A Theory of Primitive Objects

Second-Order Systems

Martín Abadi and Luca Cardelli

Digital Equipment Corporation, Systems Research Center

Abstract

We describe a second-order calculus of objects. The calculus supports object subsumption, method override, and the type Self. It is constructed as an extension of System **F** with subtyping, recursion, and first-order object types.

1. Introduction

To its founders and practitioners, object-oriented programming is a new computational paradigm distinct from ordinary procedural programming. Objects and method invocations, in their purest form, are meant to replace procedures and calls, and not simply to complement them. Is there, then, a corresponding "λ-calculus" of objects, based on primitives other than abstraction and application?

Such a specialized calculus may seem unnecessary, since untyped objects and methods can be reduced to untyped λ-terms. However, this reduction falters when we take typing and subtyping into account. It becomes even more problematic when we consider the peculiar second-order object-oriented concept of the Self type.

This paper is part of an effort to identify object calculi that are as simple and fruitful as λ-calculi. Towards this goal, we have considered untyped, first-order, and second-order systems, their equational theories, and their semantics. Here we describe, in essence, an object-oriented version of the polymorphic λ-calculus.

The starting point for this paper is a first-order calculus of objects and their types, introduced in [2]. In this calculus, an object is a collection of methods. A method is a function having a special parameter, called *self*, that denotes the same object the method belongs to. The calculus supports *object subsumption* and *method override*. Subsumption is the ability to emulate an object by means of another object that has more refined methods. Override is the operation that modifies the behavior of an object, or class, by replacing one of its methods.

We add standard second-order constructs to the first-order calculus. The resulting system is an extension of Girard's System **F** [14, 15, 22] with objects, subtyping, and recursion. Only the first-order object constructs are new. However, the interaction of second-order types, recursive types, and objects types is a prolific one. Using all these constructs, we define an interesting new quantifier, ς (sigma), similar to the μ binder of recursive types. This quantifier satisfies desirable subtyping properties that μ does not satisfy, and that are important in examples with objects. Using ς and a covariance condition we formalize the notion of Self, which is the type of the self parameter.

We take advantage of Self in some challenging examples. One is a treatment of the traditional geometric points. Another is a calculator that modifies its own methods. A third one is an object-oriented version of Scott numerals, exploiting both Self and polymorphism.

Some modern object-oriented languages support Self, subsumption, and override (e.g., [16, 23]). Correctness is obtained via rather subtle conditions, if at all. We explain Self by deriving its rules from those for more basic constructs. Thus, the problems of consistency are reduced, and hopefully clarified.

Our main emphasis is on sound typing rules for objects; because of this emphasis we restrict ourselves to a stateless computational model. However, our type theories should be largely applicable to imperative and concurrent variants of the model, and our equational theories reveal difficulties that carry over as well.

In the next section we review the first-order calculus. Section 3 concerns the second-order constructs, the quantifier ς, and matters of covariance and contravariance. In section 4 we combine the quantifier ς with object types to formalize the type Self, and we present examples. In section 5 we discuss the problem of overriding methods that return values of type Self. We conclude with a comparison with related work. The appendix lists all the typing and equational rules used in the body of the paper.

As background, we assume some familiarity with the polymorphic λ-calculus and with subtyping. We summarize our previous work on first-order systems in the next section; however, the reader may want to consult the full report on that work [Abadi, Cardelli 1994c]. A tutorial by Fisher and Mitchell [13] may also be helpful as background material.

As we mentioned above, this paper is part of a larger effort. Elsewhere, we consider denotational models [1]; we study imperative semantics [3]; we give direct rules for Self, building on the present work [3]; and we compare subtyping with the related notion of matching [4]. Palsberg has studied type inference for our first-order systems [20]; the problems of type inference remain open for second-order systems.

2. First-Order Calculi

In this section we review the typed first-order object calculus introduced in [2]. We also recall its limitations, which motivate the second-order systems that are the subject of this paper.

2.1 Informal Syntax and Semantics of Objects

We consider a minimal object calculus including *object formation*, *method invocation*, and *method override*. The calculus is very simple, with just four syntactic forms, and even without functions. It is obviously object-oriented: it has built-in objects, methods with self, and the characteristic semantics of method invocation and override. It can express object-oriented examples directly.

Syntax of the first-order ς-calculus

A,B ::=	$[l_i=B_i\ ^{i\in 1..n}]$	(l_i distinct)	types
a,b ::=			terms
	x		variable
	$[l_i=\varsigma(x_i:A)b_i\ ^{i\in 1..n}]$	(l_i distinct)	object
	a.l		field selection / method invocation
	$a.l \Leftarrow \varsigma(x:A)b$		field update / method override

Notation

- We use indexed notation of the form $\Phi_i\ ^{i\in 1..n}$ to denote sequences $\Phi_1,...,\Phi_n$.
- We use "$\triangleq$" for "equal by definition", "$\equiv$" for "syntactically identical", and "=" for "provably equal" when applied to two terms.
- [...l,l':A ...] stands for [...l:A, l':A...].
- [..., l=b, ...] stands for [..., l=ς(y:A)b, ...], for an unused y. We call l=b a *field.*

- $o.l_j:=b$ stands for $o.l_j \Leftarrow \varsigma(y:A)b$, for an unused y. We call it an *update* operation.
- We write b{x} to highlight that x may occur free in b. The substitution of a term c for the free occurrences of x in b is written $b\{x \leftarrow c\}$, or b{c} where x is clear from context.
- We identify $\varsigma(x:A)b$ with $\varsigma(y:A)(b\{x \leftarrow y\})$, for any y not occurring free in b.

An object is a collection of components $l_i=a_i$, for distinct labels l_i and associated methods a_i; the order of these components does not matter. The object containing a given method is called the method's *host* object. The symbol ς is used as a binder for the self parameter of a method; $\varsigma(x:A)b$ is a method with self parameter x of type A, to be bound to the host object, and body b.

A *field* is a degenerate method that ignores its self parameter; we talk about *field selection* and *field update*. We use the terms selection and invocation and the terms update and override somewhat interchangeably.

A method invocation is written $o.l_j$, where l_j is a label of o. It equals the result of the substitution of the host object for the self parameter in the body of the method named l_j.

A method override is written $o.l_j \Leftarrow \varsigma(y:A)b$. The intent is to replace the method named l_j of o with $\varsigma(y:A)b$; this is a single operation that involves a construction binding y in b. A method override equals a copy of the host object where the overridden method has been replaced by the overriding one. The semantics of override is functional; an override on an object produces a modified copy of the object.

An object of type $[l_i:B_i\ ^{i \in 1..n}]$ can be formed from a collection of n methods whose self parameters have type $[l_i:B_i\ ^{i \in 1..n}]$ and whose bodies have types $B_1,...,B_n$. When writing $[l_i:B_i\ ^{i \in 1..n}]$, we always assume that the l_i are distinct and that permutations do not matter. The type $[l_i:B_i\ ^{i \in 1..n}]$ exhibits only the result types B_i, and not the types of ς-bound variables. The types of all these variables is $[l_i:B_i\ ^{i \in 1..n}]$. When the method named l_i of an object of type $[l_i:B_i\ ^{i \in 1..n}]$ is invoked, it produces a result of type B_i. A method can be overridden while preserving the type of its host object. (The type of the host object cannot be allowed to change because other methods assume it, and hence soundness could be compromised.) We formalize the typing rules for objects in section 2.2.

Self-substitution is at the core of the semantics of invocation. Because of this, it is easy to define non-terminating computations without explicit use of recursion:

let $o \triangleq [l=\varsigma(x:[l:[]])x.l]$ then $o.l = x.l\{x \leftarrow o\} \equiv o.l = ...$

Using recursive types, it is possible for a method to return or to modify self. For example, let us informally assume a recursive type A equal to [l:A]; then we can write:

let $o' \triangleq [l = \varsigma(x:A)x]$ then $o'.l = x\{x \leftarrow o'\} \equiv o'$

let $o'' \triangleq [l = \varsigma(y:A)\ (y.l \Leftarrow \varsigma(x:A)x)]$ then $o''.l = (o''.l \Leftarrow \varsigma(x:A)x) = o'$

We place particular emphasis on the ability to modify self. In object-oriented languages, it is common for a method to modify field components of self. Generalizing, we allow methods to override other methods of self, or even themselves. This feature does not significantly complicate the problems that we address.

We do not provide an operation to extract a method from an object as a function. Such an operation is incompatible with object subsumption in typed calculi; in brief, the domain of the function extracted would have to be the "true type" of the object, but this type may have been forgotten by subsumption. Thus, methods are inseparable from objects and cannot be recovered as functions. This consideration inspired the use of a specialized ς-notation instead of the familiar λ-notation for parameters.

Other choices of primitives are possible; some are discussed in [2].

2.2 Object Typing and Subtyping

We now review the typing and subtyping rules for objects. Each rule has a number of antecedent judgments above a horizontal line and a single conclusion judgment below the line.

Each judgment has the form $E \vdash \mathfrak{J}$, for an environment E and an assertion $\mathfrak{J}$ depending on the judgment. An antecedent of the form "$E,E_i \vdash \mathfrak{J}_i \ \ \forall i \in 1..n$" is an abbreviation for n antecedents "$E,E_1 \vdash \mathfrak{J}_1$ ... $E,E_n \vdash \mathfrak{J}_n$" if n>0, and if n=0 for "$E \vdash \diamond$", which we read "E is well-formed". Instead, a rule containing "$j \in 1..n$" indicates that there are n separate rules, one for each j. Environments contain typing assumptions for variables; later they will also contain type-variable declarations and subtyping assumptions.

First we give rules for proving type judgments $E \vdash B$ ("B is a well-formed type in the environment E") and value judgments $E \vdash b : B$ ("b has type B in E").

(Type Object) (l_i distinct)
$$\frac{E \vdash B_i \quad \forall i \in 1..n}{E \vdash [l_i:B_i\ ^{i \in 1..n}]}$$

(Val x)
$$\frac{E, x:A, E' \vdash \diamond}{E \vdash x: A}$$

(Val Object) (where $A \equiv [l_i:B_i\ ^{i \in 1..n}]$)
$$\frac{E, x_i:A \vdash b_i : B_i \quad \forall i \in 1..n}{E \vdash [l_i=\varsigma(x_i:A)b_i\ ^{i \in 1..n}] : A}$$

(Val Select)
$$\frac{E \vdash a : [l_i:B_i\ ^{i \in 1..n}] \quad j \in 1..n}{E \vdash a.l_j : B_j}$$

(Val Override) (where $A \equiv [l_i:B_i\ ^{i \in 1..n}]$)
$$\frac{E \vdash a : A \quad E, x:A \vdash b : B_j \quad j \in 1..n}{E \vdash a.l_j \Leftarrow \varsigma(x:A)b : A}$$

A characteristic of object-oriented languages is that an object can emulate another object that has fewer methods. We call this notion *subsumption*, and say that an object can *subsume* another one. We define a particular form of subsumption that is induced by a subtyping relation between object types. An object that belongs to a given object type A also belongs to any supertype B of A, and can subsume objects in B. The judgment $E \vdash A <: B$ asserts "A is a subtype of B in environment E".

(Type Top)
$$\frac{E \vdash \diamond}{E \vdash Top}$$

(Sub Top)
$$\frac{E \vdash A}{E \vdash A <: Top}$$

(Sub Object) (l_i distinct)
$$\frac{E \vdash B_i \quad \forall i \in 1..n+m}{E \vdash [l_i:B_i\ ^{i \in 1..n+m}] <: [l_i:B_i\ ^{i \in 1..n}]}$$

(Val Subsumption)
$$\frac{E \vdash a : A \quad E \vdash A <: B}{E \vdash a : B}$$

For convenience, we add a constant, Top, a supertype of every type. The subtyping rule for objects allows a longer object type $[l_i:B_i\ ^{i \in 1..n+m}]$ to be a subtype of a shorter object type $[l_i:B_i\ ^{i \in 1..n}]$. Moreover, object types are *invariant* in their components: $[l_i:B_i\ ^{i \in 1..n+m}] <: [l_i:B_i'\ ^{i \in 1..n}]$ requires $B_i \equiv B_i'$ for $i \in 1..n$. This is necessary for soundness, basically because all components are both readable and writable.

The full first-order calculus of objects with subtyping is called $\mathbf{Ob_{1<:}}$; it can be described as a union of formal system fragments $\Delta_{\mathbf{x}} \cup \Delta_{\mathbf{K}} \cup \Delta_{\mathbf{Ob}} \cup \Delta_{<:} \cup \Delta_{<:\mathbf{Ob}}$, which are listed in appendices A and C. To facilitate comparison with other first-order calculi, $\mathbf{Ob_{1<:}}$ includes a constant type via the fragment $\Delta_{\mathbf{K}}$, but in this paper we mostly ignore constants.

2.3 Equational Theories

We associate an equational theory with $\mathbf{Ob_{1<:}}$, and with each of the calculi we study. The judgment $E \vdash b \leftrightarrow c : A$ asserts that b and c are equal as elements of A. The equational rules for $\mathbf{Ob_{1<:}}$ are $\Delta_{=} \cup \Delta_{=\mathbf{x}} \cup \Delta_{=\mathbf{Ob}} \cup \Delta_{=<:} \cup \Delta_{=<:\mathbf{Ob}}$ from appendices A and D. We give only the main rules for objects and subtyping: two rules motivated by the use of subtyping, and two evaluation rules for selection and override.

(Eq Subsumption)
$$\frac{E \vdash a \leftrightarrow a' : A \quad E \vdash A <: B}{E \vdash a \leftrightarrow a' : B}$$

(Eq Sub Object) (where $A \equiv [l_i:B_i\ ^{i\in 1..n}]$, $A' \equiv [l_i:B_i\ ^{i\in 1..n+m}]$)

$$\frac{E, x_i:A \vdash b_i : B_i \quad \forall i \in 1..n \qquad E, x_j:A' \vdash b_j : B_j \quad \forall j \in n+1..n+m}{E \vdash [l_i=\varsigma(x_i:A)b_i\ ^{i\in 1..n}] \leftrightarrow [l_i=\varsigma(x_i:A')b_i\ ^{i\in 1..n+m}] : A}$$

(Eval Select) (where $A \equiv [l_i:B_i\ ^{i\in 1..n}]$, $a \equiv [l_i=\varsigma(x_i:A')b_i\ ^{i\in 1..n+m}]$)

$$\frac{E \vdash a : A \qquad j \in 1..n}{E \vdash a.l_j \leftrightarrow b_j\{x_j \leftarrow a\} : B_j}$$

(Eval Override) (where $A \equiv [l_i:B_i\ ^{i\in 1..n}]$, $a \equiv [l_i=\varsigma(x_i:A')b_i\ ^{i\in 1..n+m}]$)

$$\frac{E \vdash a : A \qquad E, x:A \vdash b : B_j \qquad j \in 1..n}{E \vdash a.l_j \Leftarrow \varsigma(x:A)b \leftrightarrow [l_j=\varsigma(x:A')b,\ l_i=\varsigma(x_i:A')b_i\ ^{i\in (1..n+m)-\{j\}}] : A}$$

According to rule (Eq Sub Object), an object can be truncated to its externally visible methods, but only if those methods do not depend on hidden methods. The truncated object would not work otherwise.

2.4 Function Types and Recursive Types

Functions (in the form of λ-terms) can be added to $\mathbf{Ob_{1<:}}$ via standard rules, obtaining a calculus called $\mathbf{FOb_{1<:}}$. As discussed in section 3.2, functions can be encoded in terms of objects and second-order constructs.

Recursive types and values can also be added via standard rules, obtaining a calculus called $\mathbf{Ob_{1<:\mu}}$. The explicit isomorphism between μ(X)A and A{X←μ(X)A} is given by two operators called fold and unfold. Given c of type A{X←μ(X)A}, fold(A,c) has type A; conversely, given c of type μ(X)A, unfold(c) has type A{X←μ(X)A}. For example, the terms o' and o" of section 2.1 can be represented formally as follows:

let $A \triangleq \mu(X)[l:X]$
let $UA \triangleq [l:A]$
let $o' \triangleq$ fold(A,[l = ς(x:UA)fold(A,x)]) then o' has type A
let $o'' \triangleq$ fold(A,[l = ς(y:UA) (y.l⇐ς(x:UA)fold(A,x))] then o" has type A

The rules for functions and recursion are listed in appendices C and D.

2.5 The Shortcomings of First-Order Calculi

The $\mathbf{Ob_{1<:\mu}}$ calculus, consisting of objects with recursion and subtyping, is a plausible candidate as a paradigm for first-order object-oriented languages. It can be used to express many standard examples from the literature. In particular, we can write types of movable points:

$P_1 \triangleq \mu(X)[x:Int, mv\_x:Int\rightarrow X]$ one-dimensional movable points
$P_2 \triangleq \mu(X)[x,y:Int, mv\_x,mv\_y:Int\rightarrow X]$ two-dimensional movable points

We would then expect to obtain $P_2 <: P_1$, since intuitively P_2 extends P_1. But this is not provable, because the invariance of object types blocks the application of the recursive subtyping rule (Sub Rec) to the result type of mv_x.

Moreover, if we somehow allow $P_2<:P_1$, we obtain an inconsistency. Briefly, suppose that we use subsumption from $p:P_2$ to $p:P_1$, and then override the mv_x method of p with one that returns a proper element of P_1. Then, some other method of p may go wrong because it assumes that mv_x produces an element of P_2.

Hence, the failure of $P_2<:P_1$ is necessary. At the same time, it is unfortunate: in the common situation where a method returns an updated self, we lose all useful subsumption relations. In [2]

we discuss the standard solution used in object-oriented languages such as Simula-67 and Modula-3 [12, 19]. This solution sacrifices static typing information which must be recovered dynamically, thus abandoning the static typing of subsumption. This paper describes another solution that preserves static typing by taking advantage of second-order constructs.

3. Second-Order Calculi

In this section we present standard second-order extensions of first-order calculi. No new or unusual constructions are introduced. However, second-order quantifiers can be combined with recursive types to produce an interesting new concept that is recognizable as formalizing the type Self. The interaction of Self with object types is the subject of section 4.

We first introduce universal quantifiers and existential quantifiers. From existential quantifiers and recursion we define the quantifier ς. For the purpose of defining Self, only existentials and recursion are needed; one could dispense with universals. For the purposes of object-oriented languages, only the quantifier ς is needed; one could dispense with most of the second-order baggage. However, as usual, universal quantifiers may still be useful to provide polymorphism, and existential quantifiers to provide data abstraction.

We use the notation B{X} to single out the occurrences of X free in B; then B{A} stands for B{X←A} when X is clear from the context.

3.1 Universal and Existential Quantifiers

We adopt bounded universal quantifiers ∀(X<:A)B following [9, 11]. Bounded existential quantifiers ∃(X<:A)B [10] can be encoded as ∀(Y<:Top)(∀(X<:A)B{X}→Y)→Y. However, this encoding does not validate a natural and desirable η rule, called (Eval Repack<:) in appendix D. Therefore, we take bounded existentials as primitive.

We write λ(X<:A)b{X} for a bounded polymorphic function of type ∀(X<:A)B{X} that for any subtype A' of A produce a result b{A'} of type B{A'}. We write c(A') for the application of a function c:∀(X<:A)B{X} to a type A'<:A, with a result of type B{A'}.

An existentially quantified type ∃(X<:A)B{X} is the type of the pairs <A',b> where A' is a subtype of A and b is a term of type B{A'}. The type ∃(X<:A)B{X} can be seen as a partially abstract data type with interface B{X} and with representation type X known only to be a subtype of A. It is partially abstract in that it gives some information about the representation type, namely a bound. A pair <A',b> describes an element of the partially abstract data type with representation type A' and implementation b{A'} [10, 18]. In order to be fully explicit, we write the pair <A',b> more verbosely in the form:

pack X<:A=A', b{X}:B{X}

A pair c of type ∃(X<:A)B{X} can be used in the construct:

open c as X<:A,x:B{X} in d{X,x}:D

where d has access to the representation type X and implementation x, and must produce a result of a type D that does not depend on X. The requirement that X does not occur in D is necessary in order to preserve the abstraction. At evaluation time, if c is <A',b>, then the result is d{A',b}.

We assemble the following second-order calculi using fragments defined in the appendix:

$$\mathbf{F}_{<:} \triangleq \Delta_{\mathbf{x}} \cup \Delta_{\to} \cup \Delta_{<:} \cup \Delta_{<:\mathbf{X}} \cup \Delta_{<:\to} \cup \Delta_{<:\forall} \cup \Delta_{<:\exists} \qquad \mathbf{F}_{<:\mu} \triangleq \mathbf{F}_{<:} \cup \Delta_{<:\mu}$$

$$\mathbf{Ob}_{<:} \triangleq \Delta_{\mathbf{x}} \cup \Delta_{\mathbf{Ob}} \cup \Delta_{<:} \cup \Delta_{<:\mathbf{X}} \cup \Delta_{<:\mathbf{Ob}} \cup \Delta_{<:\forall} \cup \Delta_{<:\exists} \qquad \mathbf{Ob}_{<:\mu} \triangleq \mathbf{Ob}_{<:} \cup \Delta_{<:\mu}$$

$$\mathbf{FOb}_{<:} \triangleq \mathbf{F}_{<:} \cup \Delta_{\mathbf{Ob}} \cup \Delta_{<:\mathbf{Ob}} \qquad \mathbf{FOb}_{<:\mu} \triangleq \mathbf{FOb}_{<:} \cup \Delta_{<:\mu}$$

The calculi assembled in the first row are second-order calculi with function types, those in the second row have object types, and those in the third row have both. The calculi in the right column

have recursive types, while those in the left column do not. Throughout, constant types are left out because free algebras can be encoded [5].

Each of the calculi includes a set of typing fragments and a set of equational fragments. In assembling the calculi, however, we list only the typing fragments; the corresponding equational fragments are left implicit, since they can be easily identified (see appendix D). For example, calculi with the typing fragment $\Delta_{<:\mu}$ always include the equational fragment $\Delta_{=<:\mu}$.

$\mathbf{F}_{<:}$ is described in [9], but we assume only the simpler equational theory used in [11], and we add existentials. The equational rules of $\mathbf{F}_{<:}$ and its extensions are somewhat conservative. In particular, they do not equate pack terms based on different representation types. For example, we might expect that the terms (pack X<:Int=Nat, $(0,\mathrm{succ}_{\mathrm{Nat}})$) and (pack X<:Int=Int, $(0,\mathrm{succ}_{\mathrm{Int}})$) be equal at type $\exists(X<:A)X\times(X\to X)$ because they have the same behavior, but this does not seem to be derivable. This can be proved using parametricity in the theory of [21]. For simplicity we restrict attention to our conservative rules, although more ambitious rules may have advantages in combination with objects.

Universals are contravariant and existentials are covariant in their bounds, as reflected by the rules (Sub All) and (Sub Exists). Moreover, if B is covariant in X (written $B\{X^+\}$), then $\exists(X<:A)B\{X\}$ is isomorphic to $B\{A\}$. This isomorphism does not necessarily hold otherwise. In particular, $[l_i:B_i\{X\}^{\ i\in 1..n}]$ is not covariant in X even when each $B_i\{X\}$ is, and so $\exists(X<:A)[l_i:B_i\{X\}^{\ i\in 1..n}]$ is not isomorphic to $[l_i:B_i\{A\}^{\ i\in 1..n}]$. For instance, [l:X] is not covariant in X, and $\exists(X<:Int)[l:X]$ is not isomorphic to [l:Int]. The failure of this isomorphism is fairly clear since values of types $\exists(X<:Int)[l:X]$ and [l:Int] cannot be used in the same ways; for example, if o has type [l:Int] then it is legal to set o.l:=3, but there is no corresponding operation on values of type $\exists(X<:Int)[l:X]$.

In the next section we show that $\mathbf{Ob}_{<:}$ can encode $\mathbf{F}_{<:}$, and that $\mathbf{Ob}_{<:\mu}$ can encode $\mathbf{F}_{<:\mu}$ (ignoring the first-order η rule). Conversely, in [1] we show that $\mathbf{Ob}_{<:}$ can be translated into a typed λ-calculus without objects and without subtyping. In collaboration with Ramesh Viswanathan, we have since obtained a translation of $\mathbf{Ob}_{<:}$ and $\mathbf{Ob}_{<:\mu}$ into $\mathbf{F}_{<:\mu}$; this latest translation has the property of preserving subtypings. These translations of object calculi are rather complex, so we do not describe them further in this paper.

3.2 Encodings of Product and Function types

In first-order systems, object types can encode product and function types in calculi without subtyping, validating β-reductions [2]. When subtyping is added, the encodings yield invariant product and function types. We review the translation of function types. Strictly speaking, it is defined on type derivations, but we write it as a translation of type-annotated λ-terms.

Translation of invariant function types

$$\langle\!\langle A\to B\rangle\!\rangle \triangleq [\mathrm{arg}:\langle\!\langle A\rangle\!\rangle, \mathrm{val}:\langle\!\langle B\rangle\!\rangle]$$

$$\langle\!\langle x_A\rangle\!\rangle\rho \triangleq \rho(x) \qquad \rho \in \mathrm{Var}\to\mathrm{Term}$$

$$\langle\!\langle b_{A\to B}(a_A)\rangle\!\rangle\rho \triangleq (\langle\!\langle b\rangle\!\rangle\rho.\mathrm{arg} \Leftarrow \varsigma(x:\langle\!\langle A\to B\rangle\!\rangle)\,\langle\!\langle a\rangle\!\rangle\rho).\mathrm{val} \qquad \text{for } x\notin \mathrm{FV}(\langle\!\langle a\rangle\!\rangle\rho)$$

$$\langle\!\langle \lambda(x:A)b_B\rangle\!\rangle\rho \triangleq [\mathrm{arg} = \varsigma(x:\langle\!\langle A\to B\rangle\!\rangle)\,x.\mathrm{arg},\ \mathrm{val} = \varsigma(x:\langle\!\langle A\to B\rangle\!\rangle)\,\langle\!\langle b\rangle\!\rangle\rho\{x\leftarrow x.\mathrm{arg}\}]$$

In this translation, a function is mapped to an object with two methods, arg and val; the code of the function is in val, the argument is put in arg, and the function accesses it through the self variable x as x.arg. Similarly, invariant product types can be defined by $\langle\!\langle A\times B\rangle\!\rangle \triangleq [\mathrm{fst}:\langle\!\langle A\rangle\!\rangle, \mathrm{snd}:\langle\!\langle B\rangle\!\rangle]$.

These encodings yield invariant product and function types because object types are invariant in their components. At the second order, though, we have quantifiers that are variant in their bounds; combining them with object types, we can define a variant version of function types:

$$A \dot{\to} B \triangleq \forall(X<:A)\, \exists(Y<:B)\, [arg:X, val:Y]$$

We obtain $A \dot{\to} B <: A' \dot{\to} B'$ if $A' <: A$ and $B <: B'$.

This idea gives rise to an encoding of the first-order λ-calculus with subtyping but no η rule into $\mathbf{Ob}_{<:}$:

Translation of variant function types

$$\langle\!\langle A \to B \rangle\!\rangle \triangleq \forall(X<:\langle\!\langle A \rangle\!\rangle)\, \exists(Y<:\langle\!\langle B \rangle\!\rangle)\, [arg:X, val:Y] \equiv \langle\!\langle A \rangle\!\rangle \dot{\to} \langle\!\langle B \rangle\!\rangle$$

$$\langle\!\langle x_A \rangle\!\rangle_\rho \triangleq \rho(x) \qquad \rho \in Var \to Term$$

$$\langle\!\langle b_{A\to B}(a_A) \rangle\!\rangle_\rho \triangleq$$
$$\quad \text{open } \langle\!\langle b \rangle\!\rangle_\rho(\langle\!\langle A \rangle\!\rangle) \text{ as } Y<:\langle\!\langle B \rangle\!\rangle, y:[arg:\langle\!\langle A \rangle\!\rangle, val:Y]$$
$$\quad \text{in } (y.arg \Leftarrow \varsigma(x:[arg:\langle\!\langle A \rangle\!\rangle, val:Y])\, \langle\!\langle a \rangle\!\rangle_\rho).val \qquad \text{for } Y,y,x \notin FV(\langle\!\langle a \rangle\!\rangle_\rho)$$

$$\langle\!\langle \lambda(x:A) b_B \rangle\!\rangle_\rho \triangleq$$
$$\quad \lambda(X<:\langle\!\langle A \rangle\!\rangle)$$
$$\qquad (\text{pack } Y<:\langle\!\langle B \rangle\!\rangle = \langle\!\langle B \rangle\!\rangle,$$
$$\qquad\quad [arg = \varsigma(x:[arg:X, val:\langle\!\langle B \rangle\!\rangle])\, x.arg,$$
$$\qquad\quad\ val = \varsigma(x:[arg:X, val:\langle\!\langle B \rangle\!\rangle])\, \langle\!\langle b \rangle\!\rangle_{\rho\{x \leftarrow x.arg\}}]$$
$$\qquad : [arg:X, val:Y])$$

This translation can be extended trivially to recursive types and to second-order quantifiers. Hence our largest calculus, $\mathbf{FOb}_{<:\mu}$, can be embedded inside $\mathbf{Ob}_{<:\mu}$. We therefore consider $\mathbf{Ob}_{<:\mu}$ as our final pure object calculus.

Trivially, we can also obtain covariant product types, since these can be represented in terms of universal quantifiers and variant function types in $\mathbf{F}_{<:}$. A direct encoding is possible as well:

$$A \dot{\times} B \triangleq \exists(X<:A)\, \exists(Y<:B)\, [fst:X, snd:Y]$$

We obtain $A \dot{\times} B <: A' \dot{\times} B'$ if $A <: A'$ and $B <: B'$. In [8] it is shown that covariant record types $\langle l_i:A_i \ ^{i\in 1..n} \rangle$ can be represented in $\mathbf{F}_{<:}$, using covariant product types. Hence, they are also available in $\mathbf{Ob}_{<:}$.

3.3 An Encoding of Variant Object Types

Generalizing the ideas of section 3.2, we can obtain an encoding of variant object types. Through this encoding, we can make covariant any component whose method is only invoked (like val), and contravariant any component whose method is only overridden (like arg).

The idea for obtaining covariance is exactly the one used for the λ-calculus. Basically, we rewrite an object type [m:B, ...] as $\exists(Y<:B)$ [m:Y, ...]. The former is invariant in B, while the latter is covariant in B. The existential quantifier still allows the invocation of m, but blocks overrides of m from the outside, since the quantifier hides the representation type Y. Given an object o of type [m:B, ...] whose method m is never overridden from the outside, we define o' $\triangleq$ (pack Y<:B=B, o : [m:Y,...]) of type $\exists(Y<:B)$ [m:Y, ...]. We can simulate any use of o with a corresponding use of o', in particular by writing (open o' as Y<:B, x : [m:Y, ...] in x.m : B) instead of o.m.

The idea for obtaining contravariance is more complicated. (The technique used for the λ-calculus does not seem to generalize.) Given an object o of type A $\triangleq$ [m:B, ...] whose method m is never invoked from the outside, we may transform o slightly and give it a type contravariant in B; the transformation will consist in hiding m and in exhibiting a new method m_{in} that will update m internally. As a first step, we define the type A' $\triangleq \mu(X)\, \exists(Y<:(X\to B)\to X)\, [m_{in}:Y, m:B, ...]$, where

m_{in} is a new method name. Note that A' is still invariant in B. We simulate an override to the method m of o by writing the invocation $o.m_{in}(\lambda(s:A')b')$ instead of $o.m \Leftarrow \varsigma(s:A)b$, where b' imitates b. Our intent is that an invocation of m_{in} with argument $\lambda(s:A')b'$ will override m internally; therefore, the code for a typical object o' of type A' will be:

$$in([m_{in} = \varsigma(s:UA')\ \lambda(f: A' \to B)\ in(s.m \Leftarrow \varsigma(s:UA')\ f(in(s))),\ m = \varsigma(s:UA')\ s.m,\ ...\ ]): A'$$

where $UA' \triangleq [m_{in}:(A' \to B) \to A',\ m:B,\ ...]$, and for any a:UA'

$$in(a): A' \triangleq fold(A', pack\ Y<:(A' \to B) \to A' = (A' \to B) \to A',\ a : [m_{in}:Y,\ m:B,\ ...])$$

Finally we use subsumption to forget the m component, so that o' has the type:

$$\mu(X)\ \exists(Y<:(X \to B) \to X)\ [m_{in}:Y,\ ...]$$

which is contravariant in B. The method m of o' cannot be invoked from the outside since it is not even visible.

These techniques for obtaining covariance and contravariance are not fully satisfactory. For example, after making two components covariant, we are no longer able to reorder them, since $\exists(X<:A)\ \exists(Y<:B)\ C$ and $\exists(Y<:B)\ \exists(X<:A)\ C$ are not equivalent types. Therefore, we do not describe formal encodings of variant object types. Still, these techniques are suggestive, and useful in examples and in other encodings.

3.4 The Self Quantifier

Within the second-order ς-calculus with bounded quantifiers and recursion, $\mathbf{Ob}_{<:\mu}$, we can encode an interesting construction that we call the *Self quantifier*. The Self quantifier is a combination of recursion and bounded existentials, with recursion going "through the bound":

$$\varsigma(X)B \triangleq \mu(Y)\exists(X<:Y)B \qquad (Y \text{ not occurring in } B)$$

In general, any type B{A} can be transformed into a type $\exists(Y<:A)B\{Y\}$ covariant in A. (Recall from section 3.1, however, that these types are not always isomorphic.) An analogous technique applies to recursive types, and motivates our definition of the Self quantifier. Given an equation X = B{X}, we transform it into $X = \exists(Y<:X)B\{Y\}$. The solution to this equation, namely $\varsigma(X)B\{X\}$, satisfies the subtyping property:

$$\text{if } B\{X\}<:B'\{X\} \text{ then } \varsigma(X)B\{X\} <: \varsigma(X)B'\{X\},$$

even though we may not have $\mu(X)B\{X\} <: \mu(X)B'\{X\}$.

Modulo an unfolding, $\varsigma(X)B$ is the same as $\exists(X<:\varsigma(X)B)B$. Hence, by analogy with the standard interpretation of existential types, $\varsigma(X)B\{X\}$ can be understood informally as the type of pairs $\langle C, c\rangle$ consisting of a subtype C of $\varsigma(X)B\{X\}$ and an element c of B{C}.

For example, suppose that we have an element x of type $\varsigma(X)X$. Then, choosing $\varsigma(X)X$ as the required subtype of $\varsigma(X)X$, we obtain $\langle \varsigma(X)X, x\rangle : \varsigma(X)X$. Therefore we can construct:

$$\mu(x)\ \langle \varsigma(X)X, x\rangle\ :\ \varsigma(X)X$$

Less trivially, and still informally, suppose that we want to define a type M of memory cells, and to build a memory cell m:M with a read operation rd:Nat and a write operation wr:Nat→M. We can define:

$$M \triangleq \varsigma(X)[rd:Nat,\ wr:Nat \to X]$$

where the wr method should use its argument to override the rd field. For convenience, we adopt the following abbreviation to unfold a Self quantifier:

$$A\{C\} \triangleq B\{C\} \qquad \text{whenever} \qquad A \equiv \varsigma(X)B\{X\} \text{ and } C<:A$$

So, for example, $M\{M\} \equiv [rd:Nat,\ wr:Nat \to M]$.

To define a memory cell, we are going to use twice the fact that if x:M(M), then ⟨M, x⟩:M. First, we need a method body for wr that, with self s:M(M) and argument n:Nat, produces a result of type M. Since s.rd:=n has the same type as s, namely M(M), we can use ⟨M, s.rd:=n⟩:M as the body of the wr method. Therefore, we have:

$$m\colon M \triangleq \langle M,\ [rd = 0,\ wr = \varsigma(s\colon M(M))\ \lambda(n\colon Nat)\ \langle M,\ s.rd:=n\rangle]\rangle$$

Building on these intuitions, we now study the abstract properties of the Self quantifier. The two basic operations for ς are similar to the ones for existentials. One operation constructs an element of ς(X)B, given a subtype of ς(X)B and an appropriate value; it is the composition of pack for existentials and fold for recursive types. The other operation inspects an element of ς(X)B (as much as possible) and computes with its contents; it is the composition of unfold for recursive types and open for existentials.

The operation for constructing elements of type ς(X)B, in full generality, binds a type variable. Hence, we need a more complex syntax than the pairing ⟨-,-⟩ used above. We reuse the symbol ς for this second-order construct, in the same way we use λ for both first-order and second-order binders. We refine the notation ⟨C,b⟩ to ς(Y<:A=C)b. The term ς(Y<:A=C)b binds C to Y in b, and requires C to be a subtype of A. Within b, the types Y and C are equivalent. The type of the whole term is A.

We define, for A≡ς(X)B{X}, C<:A, and b{C}:B{C}:

$$\varsigma(Y<:A=C)\ b\{Y\} \triangleq \mathrm{fold}(A,\ (\mathrm{pack}\ Y<:A=C,\ b\{Y\}:B\{Y\}))$$

and, for c:A and d{Y,y}:D, where Y does not occur in D:

$$(\mathrm{use}\ c\ \mathrm{as}\ Y<:A,\ y:B\{Y\}\ \mathrm{in}\ d\{Y,y\}:D) \triangleq (\mathrm{open}\ \mathrm{unfold}(c)\ \mathrm{as}\ Y<:A,\ y:B\{Y\}\ \mathrm{in}\ d\{Y,y\}:D)$$

The following rules for the Self quantifier can be derived from the rules for μ and ∃.

Δ_{ς}

(Type Self)
$$\frac{E,\ X<:Top \vdash B}{E \vdash \varsigma(X)B}$$

(Sub Self)
$$\frac{E,\ X<:Top \vdash B <: B'}{E \vdash \varsigma(X)B <: \varsigma(X)B'}$$

(Val Self) (where $A \equiv \varsigma(X)B\{X\}$)
$$\frac{E \vdash C <: A \qquad E \vdash b\{C\} : B\{C\}}{E \vdash \varsigma(Y<:A=C)b\{Y\} : A}$$

(Val Use) (where $A \equiv \varsigma(X)B\{X\}$)
$$\frac{E \vdash c : A \qquad E \vdash D \qquad E,\ Y<:A,\ y:B\{Y\} \vdash d : D}{E \vdash (\mathrm{use}\ c\ \mathrm{as}\ Y<:A,\ y:B\{Y\}\ \mathrm{in}\ d:D) : D}$$

$\Delta_{=\varsigma}$

(Eq Self) (where $A \equiv \varsigma(X)B\{X\},\ A' \equiv \varsigma(X)B'\{X\}$)
$$\frac{E \vdash C <: A' \qquad E \vdash A \qquad E,\ X<:Top \vdash B'\{X\} <: B\{X\} \qquad E \vdash b\{C\} \leftrightarrow b'\{C\} : B'\{C\}}{E \vdash \varsigma(Y<:A=C)b\{Y\} \ \leftrightarrow\ \varsigma(Y<:A'=C)b'\{Y\} : A}$$

(Eq Use) (where $A \equiv \varsigma(X)B\{X\}$)

$$\frac{E \vdash c \leftrightarrow c' : A \qquad E \vdash D \qquad E,\ Y<:A,\ y:B\{Y\} \vdash d \leftrightarrow d' : D}{E \vdash (\text{use } c \text{ as } Y<:A,\ y:B\{Y\} \text{ in } d:D) \leftrightarrow (\text{use } c' \text{ as } Y<:A,\ y:B\{Y\} \text{ in } d':D) : D}$$

(Eval Unself) (where $A \equiv \varsigma(X)B\{X\}$, $c \equiv \varsigma(Z<:A=C)b\{Z\}$)

$$\frac{E \vdash c : A \qquad E \vdash D \qquad E,\ Y<:A,\ y:B\{Y\} \vdash d\{Y,y\} : D}{E \vdash (\text{use } c \text{ as } Y<:A, y:B\{Y\} \text{ in } d\{Y,y\}:D) \leftrightarrow d\{C,b\{C\}\} : D}$$

(Eval Reself) (where $A \equiv \varsigma(X)B\{X\}$)

$$\frac{E \vdash b : A \qquad E,\ y:A \vdash d\{y\} : D}{E \vdash (\text{use } b \text{ as } Y<:A, y:B\{Y\} \text{ in } d\{\varsigma(Y'<:A=Y)y\} :D) \leftrightarrow d\{b\} : D}$$

Notation We write:

- $\varsigma\langle A,c\rangle$ for $\varsigma(X<:A=A)c$ when X does not occur in c
- $\varsigma(X=A)c\{X\}$ for $\varsigma(X<:A=A)c\{X\}$

These rules are roughly analogous to corresponding rules for existential types and for recursive types. The rules in Δ_{ς} serve for proving subtypings between types of the form $\varsigma(X)B$, and for typing terms with the constructs ς and use. The rules in $\Delta_{=\varsigma}$ include two congruence rules ((Eq Self) and (Eq Use)) and two evaluation rules ((Eval Unself) and (Eval Reself)) for the constructs ς and use.

Note in particular the expected rule of covariance, (Sub Self). This rule holds because existential types are covariant in their bounds, and because recursion in $\mu(Y)\exists(X<:Y)B\{X\}$ involves only a single covariant position. It can be verified as follows:

$E,X<:Top \vdash B <: B'$

$\Rightarrow E,Z<:Top,Y<:Z,X<:Y \vdash B <: B'$ by standard weakening lemmas, for fresh Y,Z

$\Rightarrow E,Z<:Top,Y<:Z \vdash \exists(X<:Y)B <: \exists(X<:Z)B'$ by (Sub Exists)

$\Rightarrow E \vdash \mu(Y)\exists(X<:Y)B <: \mu(Z)\exists(X<:Z)B'$ by (Sub Rec)

The memory cell definition can now be understood formally, with the $\varsigma\langle M,...\rangle$ notation replacing the informal notation $\langle M,...\rangle$:

$M \triangleq \varsigma(Self)[rd:Nat,\ wr:Nat \rightarrow Self]$

$m: M \triangleq \varsigma(Self=M)[rd = 0,\ wr = \varsigma(s:M(Self))\ \lambda(n:Nat)\ \varsigma\langle Self,\ s.rd:=n\rangle]$

Later examples frequently adopt this choice of Self as a bound variable.

3.5 A Pure Second-Order Object Calculus

We conclude this section by considering a pure second-order object calculus, **ςOb**, based exclusively on object types and the Self quantifier (see appendix E):

$$\textbf{ςOb} \triangleq \Delta_{\mathbf{x}} \cup \Delta_{\mathbf{Ob}} \cup \Delta_{<:} \cup \Delta_{<:\mathbf{X}} \cup \Delta_{<:\mathbf{Ob}} \cup \Delta_{\varsigma}$$

This calculus departs from second-order λ-calculi by omitting function types and the standard quantifiers. It seems that, in **ςOb**, the only function types that can be encoded are invariant, and that the standard second-order quantifiers are not expressible. Still, as the next section demonstrates, the Self quantifier provides an essential second-order feature of object calculi, namely the type Self, with a form of type recursion. Interestingly, then, **ςOb** is a small second-order object calculus that covers a spectrum of object-oriented notions.

4. Object Types with Self

The payoff of the Self quantifier comes when it is used in conjunction with object types. Object types with Self are obtained by the combination of the simple object types of section 2.2 with the Self quantifier of section 3.4. These new types allow subsumption between objects containing methods that return self.

In this section, we first derive the rules for the combination of objects with Self. We then show how these derived rules can easily provide typings for some interesting examples. We work entirely within $\mathbf{Ob}_{<:\mu}$, that is, within the second-order calculus with bounded quantifiers, recursion, and simple object types.

4.1 ς-Objects

In this section we examine types of the form:

$\varsigma(X)[l_i:B_i\{X\}^{\,i\in 1..n}]$ where X occurs only covariantly in each $B_i\{X\}$

We call these structures ς-object types, and ς-objects their corresponding values. The parameter X in $\varsigma(X)[l_i:B_i\{X\}^{\,i\in 1..n}]$ is intended as the Self type. Note that we do not require that $[l_i:B_i\{X\}^{\,i\in 1..n}]$ be covariant in X, only that each $B_i\{X\}$ be covariant in X.

Although ς-object types are obtained by applying a Self quantifier (which has no covariance restrictions) to an object type, for the most part we consider $\varsigma(X)[l_i:B_i\{X\}^{\,i\in 1..n}]$ as a single type construction. The covariance requirement is necessary when selecting components of ς-objects. For emphasis, we use a special syntax for the combination of Self quantifiers, object types, and the covariance requirement:

$\varsigma(X^+)[l_i:B_i\{X\}^{\,i\in 1..n}] \triangleq \varsigma(X)[l_i:B_i\{X\}^{\,i\in 1..n}]$ when X occurs only covariantly in each $B_i\{X\}$

The covariance requirement implies that X_i must not occur within any object type within B_i, since object types are invariant in their components. For example, $\varsigma(X)[l: \varsigma(Y)[m:X, n:Y]]$ violates the covariance requirement. Hence, informally, we may say that Self types do not nest: there is a single meaningful Self type within each pair of object brackets.

It is often useful to consider an unfolding $A\langle C\rangle$ of a ς-object type A:

$A\langle C\rangle \triangleq [l_i:B_i\{C\}^{\,i\in 1..n}]$ whenever $A \equiv \varsigma(X^+)[l_i:B_i\{X\}^{\,i\in 1..n}]$ and $C<:A$

We frequently consider $A\langle X\rangle$, for a variable X, and the *self-unfolding* $A\langle A\rangle$ of A. (When building an element of type A it is common to build first an element of type $A\langle A\rangle$.) We say that a type $C<:A$ and an element of $A\langle C\rangle$ constitute an *implementation* of A, since they can be used to build an element of A. Then C is the *representation type* for the implementation.

The operations on ς-objects are defined as follows. Assume that a has type A with $A\equiv \varsigma(X^+)[l_i:B_i\{X\}^{\,i\in 1..n}]$ and $A\langle X\rangle\equiv[l_i:B_i\{X\}^{\,i\in 1..n}]$, and set, with some overloading of notation:

$a.l_j \triangleq$
(use a as $Z<:A$, $y:A\langle Z\rangle$ in $y.l_j : B_i\{A\}$)

$a.l_j \Leftarrow \varsigma(Y<:A, y:A\langle Y\rangle, x:A\langle Y\rangle)b\{Y,y,x\} \triangleq$
(use a as $Z<:A$, $y:A\langle Z\rangle$ in $\varsigma(Y<:A=Z)\ (y.l_j \Leftarrow \varsigma(x:A\langle Y\rangle)b\{Y,y,x\}) : A$)

The ς-object selection operation reduces fairly simply to a regular selection operation on the underlying object.

The ς-object override operation is more interesting, although similarly it reduces to an override operation on the underlying object. The overriding method b can take advantage of three variables: (1) $Y<:A$, the unknown subtype of A that was used to construct a; (2) $y:A\langle Y\rangle$, the raw object inside a, which can be thought of as the *old self*; y is the value of self at the time the

overriding takes place, containing the old version of method l_j; (3) x:A(Y), the regular self of the overriding method b. From these variables, b must produce a result of type $B_j\{Y\}$, parametrically in Y.

Combining the rules for objects and for Self quantification with the definitions above, we derive the following rules. (The complete set is given in appendix B.)

(Type ς Object)
$$\frac{E, X<:Top \vdash B_i\{X^+\} \quad \forall i \in 1..n}{E \vdash \varsigma(X^+)[l_i:B_i\{X\}^{\,i\in 1..n}]}$$

(Sub ς Object) (l_i distinct)
$$\frac{E, X<:Top \vdash B_i\{X^+\} \quad \forall i \in 1..n+m}{E \vdash \varsigma(X^+)[l_i:B_i\{X\}^{\,i\in 1..n+m}] <: \varsigma(X^+)[l_i:B_i\{X\}^{\,i\in 1..n}]}$$

(Val ς Object) (where $A \equiv \varsigma(X^+)[l_i:B_i\{X\}^{\,i\in 1..n}]$)
$$\frac{E \vdash C<:A \quad E \vdash b\{C\} : A(C)}{E \vdash \varsigma(Y<:A=C)\, b\{Y\} : A}$$

(Val ς Select) (where $A \equiv \varsigma(X^+)[l_i:B_i\{X\}^{\,i\in 1..n}]$)
$$\frac{E \vdash a : A \quad j \in 1..n}{E \vdash a.l_j : B_j\{A\}}$$

(Val ς Override) (where $A \equiv \varsigma(X^+)[l_i:B_i\{X\}^{\,i\in 1..n}]$)
$$\frac{E \vdash a : A \quad E, Y<:A, y:A(Y), x:A(Y) \vdash b : B_j\{Y^+\} \quad j \in 1..n}{E \vdash a.l_j \Leftarrow \varsigma(Y<:A, y:A(Y), x:A(Y))b : A}$$

(Eq ς Object) (where $A \equiv \varsigma(X^+)[l_i:B_i\{X\}^{\,i\in 1..n}]$, $A' \equiv \varsigma(X^+)[l_i:B_i\{X\}^{\,i\in 1..n+m}]$)
$$\frac{E \vdash C<:A' \quad E \vdash b\{C\} \leftrightarrow b'\{C\} : A'(C)}{E \vdash \varsigma(Y<:A=C)b\{Y\} \;\leftrightarrow\; \varsigma(Y<:A'=C)b'\{Y\} : A}$$

(Eq Sub ς Object) (where $A \equiv \varsigma(X^+)[l_i:B_i\{X\}^{\,i\in 1..n}]$, $A' \equiv \varsigma(X^+)[l_i:B_i\{X\}^{\,i\in 1..n+m}]$)
$$\frac{E \vdash C<:A' \quad E, x_i:A(C) \vdash b_i\{C\} : B_i\{C\}\ \forall i \in 1..n \quad E, x_j:A'(C) \vdash b_j\{C\} : B_j\{C\}\ \forall j \in n+1..n+m}{E \vdash \varsigma(Y<:A=C)[l_i=\varsigma(x_i:A(Y))b_i\{Y\}^{\,i\in 1..n}] \;\leftrightarrow\; \varsigma(Y<:A'=C)[l_i=\varsigma(x_i:A'(Y))b_i\{Y\}^{\,i\in 1..n+m}] : A}$$

(Eval ς Select) (where $A \equiv \varsigma(X^+)[l_i:B_i\{X\}^{\,i\in 1..n}]$, $c \equiv \varsigma(Z<:A=C)a\{Z\}$)
$$\frac{E \vdash c : A \quad j \in 1..n}{E \vdash c.l_j \;\leftrightarrow\; a\{C\}.l_j : B_j\{A\}}$$

(Eval ς Override) (where $A \equiv \varsigma(X^+)[l_i:B_i\{X\}^{\,i\in 1..n}]$, $c \equiv \varsigma(Z<:A=C)a\{Z\}$)
$$\frac{E \vdash c : A \quad E, Y<:A, y:A(Y), x:A(Y) \vdash b\{Y,y,x\} : B_j\{Y^+\} \quad j \in 1..n}{E \vdash c.l_j \Leftarrow \varsigma(Y<:A,y:A(Y),x:A(Y))b\{Y,y,x\} \leftrightarrow \varsigma(Z<:A=C)\, a\{Z\}.l_j \Leftarrow \varsigma(x:A(Z))b\{Z,a\{Z\},x\} : A}$$

The most remarkable fact is that the (Sub ς Object) rule holds for ς-object types. We recall that in section 2.5 we found that the analogous rule for recursive object types did not hold.

The (Val ς Object) rule can be used to build a ς-object $\varsigma(Y<:A=C)b\{Y\}$ from a subtype C of the desired ς-object type A, and from a regular object b{C}. The Y variable in b{Y} is the Self type, in case the methods of b need to refer to it.

It is easy to define ς-objects. When building a ς-object by (Val ς Object), its methods are not required to operate on an arbitrary self: they just need to match the given representation type of the object being constructed. That is, to construct a ς-object of type $A \equiv \varsigma(X^+)[l_i:B_i\{X\}^{\,i\in 1..n}]$ we need

only a set of methods b_j:$B_j\{C\}$ for some C<:A (not b_j:$B_j\{X\}$ for an arbitrary X<:A). We often let C equal A, and we seldom let C equal a type variable. Moreover, each of these methods can assume the existence a self parameter x_j:$[l_i:B_i\{C\}\ ^{i\in 1..n}]$. (See the rules (Val ςObject) and (Val Object).) We rarely need to work parametrically with an arbitrary X<:A. However, the flexibility of using an arbitrary subtype of A is critical in the derivation of (Val ςOverride). In section 5.1 we will see that this flexibility has a price.

The (Eq Sub ςObject) rule is of limited power because the same C appears on both sides of the conclusion. We can trace back this limitation to a similar one in the rules for existentials, which was discussed in section 3.1.

We now verify some of these rules in detail.

- (Val ςSelect) (where $A\equiv\varsigma(X^+)[l_i:B_i\{X\}\ ^{i\in 1..n}]$)
  The derivation relies on covariance of B_i in X, and subsumption.
  E, Y<:A, y:A⟨Y⟩ ⊢ $y.l_j : B_j\{Y\}$ by (Val Select)
  E, Y<:A, y:A⟨Y⟩ ⊢ $B_j\{Y\} <: B_j\{A\}$ since Y<:A and $B_j\{Y\}$ is covariant in Y
  E, Y<:A, y:A⟨Y⟩ ⊢ $y.l_j : B_j\{A\}$ by (Val Subsumption)
  E ⊢ (use a as Y<:A, y:A⟨Y⟩ in $y.l_j :B_j\{A\}$) : $B_j\{A\}$ by (Val Use) since E ⊢ a:A

- (Val ςOverride) (where $A\equiv\varsigma(X^+)[l_i:B_i\{X\}\ ^{i\in 1..n}]$)
  The derivation relies on the full power of (Val Self). Note that y:A⟨Y⟩ does not imply y:A⟨A⟩, even though Y<:A. Otherwise b would only need to have type $B_i\{A\}$, and this would permit unsound overrides.
  E, Y<:A, y:A⟨Y⟩, x:A⟨Y⟩ ⊢ $b\{Y,y,x\} : B_j\{Y^+\}$ by assumption
  E, Y<:A, y:A⟨Y⟩ ⊢ $y.l_j \Leftarrow \varsigma(x:A⟨Y⟩)b\{Y,y,x\} : A⟨Y⟩$ by (Val Override)
  E, Y<:A, y:A⟨Y⟩ ⊢ $\varsigma(Z<:A=Y)\ y.l_j \Leftarrow \varsigma(x:A⟨Z⟩)b\{Z,y,x\} : A$ by (Val Self)
  E ⊢ (use a as Y<:A, y:A⟨Y⟩ in $\varsigma(Z<:A=Y)\ y.l_j \Leftarrow \varsigma(x:A⟨Z⟩)b\{Z,y,x\}$:A) : A by (Val Use)

- (Eval ςSelect) (where $A\equiv\varsigma(X^+)[l_i:B_i\{X\}\ ^{i\in 1..n}]$, $c\equiv\varsigma(Z<:A=C)a\{Z\}$)
  E, Y<:A, y:A⟨Y⟩ ⊢ $y.l_j : B_j\{A\}$ as for (Val ςSelect)
  E ⊢ (use a as Y<:A, y:A⟨Y⟩ in $y.l_j$:$B_j\{A\}$) ↔ $a\{C\}.l_j : B_j\{A\}$ by (Eval Unself)

- (Eval ςOverride) (where $A\equiv\varsigma(X^+)[l_i:B_i\{X\}\ ^{i\in 1..n}]$, $c\equiv\varsigma(Z<:A=C)a\{Z\}$)
  E, Y<:A, y:A⟨Y⟩ ⊢ $\varsigma(Z<:A=Y)\ y.l_j \Leftarrow \varsigma(x:A⟨Z⟩)b\{Z,y,x\} : A$ as for (Val ςOverride)
  E ⊢ (use c as Y<:A, y:A⟨Y⟩ in $\varsigma(Z<:A=Y)\ y.l_j \Leftarrow \varsigma(x:A⟨Z⟩)b\{Z,y,x\}$:A)
  ↔ $\varsigma(Z<:A=C)\ a\{C\}.l_j \Leftarrow \varsigma(x:A⟨Z⟩)b\{Z,a\{C\},x\}$ by (Eval Unself)
  ↔ $\varsigma(Z<:A=C)\ a\{Z\}.l_j \Leftarrow \varsigma(x:A⟨Z⟩)b\{Z,a\{Z\},x\}$: A by (Eq Self)

4.2 Examples

We are now ready to examine some object-oriented examples (cf. [2]). We find that these examples can be typed rather easily when seen in terms of ς-objects, even when a method needs to return or to modify self. The main benefit of using ς-object types, rather than recursive types, is that we obtain useful subtypings from the rule (Sub ςObject).

4.2.1 Movable Points

This is a modified version of the problematic example of section 2.5, obtained by replacing μ with ς. We define the types of one-dimensional and two-dimensional movable points:

$P_1 \triangleq \varsigma(Self^+)[x:Int, mv\_x:Int\to Self]$
$P_2 \triangleq \varsigma(Self^+)[x,y:Int, mv\_x,mv\_y:Int\to Self]$

We have the desirable property $P_2 <: P_1$, by (Sub ς Object).

Next we define the one-dimensional origin point, where Self is P_1. Recall that $\varsigma\langle A,c\rangle$ abbreviates $\varsigma(X<:A=A)c$ for an unused X.

$$\text{origin}_1 : P_1 \triangleq \varsigma(\text{Self}=P_1)[x=0, \text{mv\_x}=\varsigma(s:P_1\{\text{Self}\})\lambda(dx:\text{Int})\varsigma\langle \text{Self}, s.x:=s.x+dx\rangle]$$

The typing $\text{origin}_1:P_1$ can be derived as follows, with $P_1\{P_1\}\equiv[x:\text{Int}, \text{mv\_x}:\text{Int}\to P_1]$ as the chosen representation type for P_1:

$s:P_1\{P_1\}, dx:\text{Int} \vdash s.x:=s.x+dx : P_1\{P_1\}$ by (Val Select), (Val Override)

$s:P_1\{P_1\}, dx:\text{Int} \vdash \varsigma\langle P_1, s.x:=s.x+dx\rangle: P_1$ by (Val ς Object)

$s:P_1\{P_1\} \vdash \lambda(dx:\text{Int})\varsigma\langle P_1, s.x:=s.x+dx\rangle: \text{Int}\to P_1$ by (Val Fun)

$\vdash [x=0, \text{mv\_x}=\varsigma(s:P_1\{P_1\})\lambda(dx:\text{Int})\varsigma\langle P_1, s.x:=s.x+dx\rangle] : P_1\{P_1\}$ by (Val Object)

$\vdash \varsigma(\text{Self}=P_1)[x=0, \text{mv\_x}=\varsigma(s:P_1\{\text{Self}\})\lambda(dx:\text{Int})\varsigma\langle \text{Self}, s.x:=s.x+dx\rangle] : P_1$ by (Val ς Object)

The rule (Val ς Select) allows us to invoke methods whose type involves Self:

$$\text{origin}_1.\text{mv\_x} : \text{Int}\to P_1$$

Moreover, the equational theory allows us to derive expected equivalencies, such as:

$$\text{origin}_1.\text{mv\_x}(1) \leftrightarrow \varsigma(\text{Self}=P_1)[x=1, \text{mv\_x}=\varsigma(s:P_1\{\text{Self}\})\lambda(dx:\text{Int})\varsigma\langle \text{Self}, s.x:=s.x+dx\rangle] : P_1$$

that is, the unit point equals the result of moving the origin point. In light of these properties, we consider that this treatment of movable points is satisfactory.

We take advantage of this example to comment on the so-called *binary methods*, as an aside. In object-oriented programming, binary methods have proven generally problematic [7]. As we will see, we do not have much to contribute on this subject. A typical example of a binary method is an equality method in a point object:

$$P_{1eq} \triangleq \varsigma(\text{Self})[x:\text{Int}, \text{mv\_x}:\text{Int}\to\text{Self}, \text{eq}:\text{Self}\to\text{Bool}]$$
$$\text{origin}_{1eq} \triangleq \varsigma(\text{Self}=P_{1eq})[x=0, \text{mv\_x}=..., \text{eq} = \varsigma(s:P_{1eq}\{\text{Self}\})\ \lambda(p:\text{Self})\ s.x =_{\text{Int}} p.x]$$

Binary methods violate the covariance requirement of ς-object types. We shall temporarily ignore this requirement, which is just a convention. We use the rules for the ς quantifier, which do not depend on covariance, instead of the derived rules for ς-objects.

We discover that binary methods cannot be invoked effectively because of typing restrictions. Expanding the encoding of method invocation, we may try to pull out the eq method from a ς-object p of type P_{1eq}:

use p as $\text{Self}<:P_{1eq}$, $z:[x:\text{Int}, \text{mv\_x}:\text{Int}\to\text{Self}, \text{eq}:\text{Self}\to\text{Bool}]$ in z.eq (not typable)

The rule (Val Use) requires that z.eq be given a type independent of Self. This is not possible because Self is in contravariant position in Self→Bool. Although we cannot pull out the eq method from p, we can still apply it within the scope of the use construct, if we can find an adequate argument. One possibility is:

use p as $\text{Self}<:P_{1eq}$, $z:[x:\text{Int}, \text{mv\_x}:\text{Int}\to\text{Self}, \text{eq}:\text{Self}\to\text{Bool}]$ in $z.\text{eq}(z.\text{mv\_x}(1))$: Bool

which compares p with its translation by 1, returning false. However, most advantageous uses of eq would involve comparisons with independently obtained points, and these comparisons are not possible. This situation arises, essentially, because independent instances of the same existential type do not intermix.

In order to avoid the contravariant occurrence of Self, we may define:

$$P_{1eq\_co} \triangleq \varsigma(Self^+)[x:Int, mv\_x:Int\to Self, eq:P_1\to Bool]$$
$$origin_{1eq\_co} \triangleq \varsigma(Self=P_{1eq\_co})[x=0, mv\_x=..., eq = \varsigma(s:P_{1eq\_co}(Self))\ \lambda(p:P_1)\ s.x =_{Int} p.x]$$

These definitions are more useful than the previous ones. In particular, $origin_{1eq\_co}.eq(p)$ is well typed whenever p has type P_1. However, eq is no longer a binary method, since its argument does not have the same type as self.

4.2.2 Object-Oriented Natural Numbers

The type of Scott numerals [24] has an object-oriented counterpart:

$$N_{Ob} \triangleq \varsigma(Self^+)[succ:Self, case:\forall(Z<:Top)Z\to(Self\to Z)\to Z]$$

This type is well-formed because $\forall(Z<:Top)Z\to(X\to Z)\to Z$ is covariant in X.

Informally, an object of type N_{Ob} represents a number. The succ method of a number returns its successor. Given a type Z, a value z of type Z, and a function f from N_{Ob} to Z, the case method returns z if the number is 0, and f(n) if the number is n+1.

The zero numeral can be defined as:

$$zero_{Ob} : N_{Ob} \triangleq$$
$$\varsigma(Self=N_{Ob})$$
$$[case = \lambda(Z<:Top)\ \lambda(z:Z)\ \lambda(f:Self\to Z)\ z,$$
$$succ = \varsigma(n:N_{Ob}(Self))\ \varsigma\langle Self, n.case := \lambda(Z<:Top)\ \lambda(z:Z)\ \lambda(f:Self\to Z)\ f(\varsigma\langle Self,n\rangle)\rangle]$$

The other numerals can be obtained from $zero_{Ob}$ by invoking succ repeatedly. Some familiar operations are expressible:

$$succ : N_{Ob}\to N_{Ob} \triangleq \lambda(n:N_{Ob})\ n.succ$$
$$pred : N_{Ob}\to N_{Ob} \triangleq \lambda(n:N_{Ob})\ n.case(N_{Ob})(zero_{Ob})(\lambda(p:N_{Ob})p)$$
$$iszero : N_{Ob}\to Bool \triangleq \lambda(n:N_{Ob})\ n.case(Bool)(true)(\lambda(p:N_{Ob})false)$$

4.2.3 A Calculator

Our final example is that of a calculator object. We exploit the ability to override methods to record the pending arithmetic operation. When an operation add or sub is entered, the equals method is overridden with code for addition or subtraction. The first two components (arg, acc) are needed for the internal operation of the calculator, while the other four (enter, add, sub, equals) provide the user interface.

$$C = \varsigma(Self^+)[arg,acc: Real, enter: Real\to Self, add,sub: Self, equals: Real]$$

By subsumption, the calculator also has type:

$$Calc = \varsigma(Self^+)[enter: Real\to Self, add,sub: Self, equals: Real]$$

This shorter type is the one shown to users of the calculator, and is a supertype of C. We define the calculator as follows:

```
calculator: Calc  ≜
  ς(Self=C)
    [arg = 0.0,
     acc = 0.0,
     enter = ς(s:C(Self)) λ(n:Real) ς⟨Self, s.arg := n⟩,
     add = ς(s:C(Self)) ς⟨Self, (s.acc := s.equals).equals ⇐ ς(s':C(Self)) s'.acc+s'.arg⟩,
     sub = ς(s:C(Self)) ς⟨Self, (s.acc := s.equals).equals ⇐ ς(s':C(Self)) s'.acc-s'.arg⟩,
     equals = ς(s:C(Self)) s.arg]
```

This definition is meant to provide the following behavior:

```
calculator .enter(5.0) .equals                  =   5.0
calculator .enter(5.0) .sub .enter(3.5) .equals =   1.5
calculator .enter(5.0) .add .add .equals        =  15.0
```

A scientific calculator can also be defined, with additional state and operations. Its inner design could be quite different from that of our basic calculator, but the scientific calculator's type may still be a subtype of Calc.

5. Overriding and Self

In this section we discuss attempts to override methods that return self. If we want to override a method of a ς-object of type A, the new method must work for any possible subtype of A. This is because the ς-object might have been constructed as an element of an unknown proper subtype of A. If the new method returns self, it is critical that the type of its result be the unknown subtype of A, because one of the other methods may be invoked on the result. We say that the overriding method must be "parametric in self "; this turns out to be a difficult criterion to meet.

It should not be too surprising that it is hard to override methods that return self. After all, the technique for obtaining ς-object subtypings is based on that of section 3.3 for obtaining covariant object components, which cannot be usefully overridden.

5.1 Overriding from the Outside

In section 4.1 we explain that it is easy to create a ς-object, because its initial methods needs to work only for the actual type of the object being constructed. In particular, methods that override self present no difficulties. However, if we want to override a method of an existing ς-object o:A, the new method must work for any possible B<:A, because o might have been built as an element of B. We do not know either the "true type" of o, or the "true type" of the self parameters of its methods. When overriding a method of o, the overriding method can assume only that the object has been constructed from an unknown Self<:A. The same difficulty would likely surface at object-creation time, if we were creating objects incrementally, adding methods to an empty object, instead of creating full objects at once.

This is where we need the complex derived rule for overriding ς-objects, (Val ςOverride). Consider, for example, the type:

$$Q \triangleq \varsigma(Self^{+})[n,f:Int,\ m:Self] \qquad \text{with} \qquad Q(X) \equiv [n,f:Int,\ m:X]$$

An overriding method for o≡ς(Y<:Q=C)b{Y} can use in its body the variables Self<:Q, x':Q(Self), and x:Q(Self), where x' is in fact b{C}, according to (Eval ςOverride), and x is the self of the new method. We can therefore override the f method of Q with any of the following method bodies:

o.f ⇐ ς(Self<:Q, x':Q{Self}, x:Q{Self}) *(any of the following:)*

x'.n + 1	setting o.f to produce b.n + 1 (constantly)
x.n + 1	setting o.f to produce 1 + the value of n when f is invoked

Let us now attempt to override the m method. The typing rule (Val ςOverride) requires that, from the variables Self<:Q, x':Q{Self}, x:Q{Self} at its disposal, the overriding method must produce a value of type Self. Here are some possibilities:

o.m ⇐ ς(Self<:Q, x':Q{Self}, x:Q{Self}) *(any of the following:)*

x'.m	setting o.m to produce the current b.m (constantly)
x.m	setting o.m to diverge

However, we cannot override o.m with anything useful. Note, first, that we cannot synthesize a value of type Self from scratch. Second, we cannot return x' or x, nor ς⟨Q,x'⟩ or ς⟨Q,x⟩, because none of these can be given type Self. Third, any update to x', x, ς⟨Q,x'⟩, or ς⟨Q,x⟩ will preserve their original type, so we cannot return the updated terms either. Finally, ς⟨Self,x⟩:Self is not derivable, for an unknown Self<:Q.

Moreover, it would be unsound to ignore these typing problems and return, say, ς⟨Self,x⟩ or ς⟨Self,x'⟩. The reason can be seen in the following example, which builds a ς-object r_2 from a proper subtype R_1 of its own type R_2:

$R_2 \triangleq \varsigma(Self^+)[p:Self,\ q:Int]$
$R_1 <: R_2 \triangleq \varsigma(Self^+)[p:Self,\ q:Int,\ t:Int]$
$r_1 : R_1 \triangleq \varsigma(Y<:R_1=R_1)[p=\varsigma(s:R_1\{Y\}))\varsigma\langle Y,s\rangle,\ q=0,\ t=0]$
$r_2 : R_2 \triangleq \varsigma(Y<:R_2=R_1)[p=r_1,\ q=\varsigma(s:R_2\{Y\})s.p.t]$

$r_2.p \Leftarrow \varsigma(Self<:R_2, x':R_2\{Self\}, x:R_2\{Self\})\ \varsigma\langle Self,x\rangle$
$\quad = \varsigma(Y<:R_2=R_1)[p=\varsigma(x:R_2\{Y\})\varsigma\langle Y,x\rangle,\ \ q=\varsigma(s:R_2\{Y\})\ s.p.t]$ (unsound)
$r_2.p \Leftarrow \varsigma(Self<:A, x':A\{Self\}, x:A\{Self\})\ \varsigma\langle Self,x'\rangle$
$\quad = \varsigma(Y<:R_2=R_1)[p=\varsigma(x:A\{Y\})r_2,\ \ q=\varsigma(s:R_2\{Y\})\ s.p.t]$ (unsound)

Unsound behavior can be observed by invoking q after either of the two updates above, because the field t is missing from s.p.

The reason for this unsound behavior can be traced back to the rule for constructing ς-objects. Because of the flexibility we have in constructing ς-objects out of proper subtypes of themselves, the x and x' parameters at our disposal when overriding may be shorter than the subtype used originally when constructing the object. For example, r_2 is constructed with a component p that is longer than the body of r_2, to which x or x' would be bound. Therefore, returning x or x' would not be safe, because they would be too short.

In conclusion, we discover that the (Val ςOverride) rule, although very powerful for overriding simple methods and fields, is not sufficient to allow us to override methods that return a value of type Self. One solution to this problem is discussed in the next section.

5.2 Recoup

In this section we introduce a special method called *recoup* with an associated run-time invariant. Recoup is a method that returns self immediately. The invariant asserts that the result of recoup is its host object.

Let us redefine the type Q of section 5.1, by adding a method called recoup:

Q ≜ ς(Self⁺)[recoup: Self, n,f:Int, m:Self]

We can build an element of Q as follows:

$$o : Q \triangleq \varsigma(Self=Q)\, b, \quad \text{where} \quad b \equiv [recoup=\varsigma(s:Q(Self))\varsigma\langle Self,s\rangle, n=..., f=..., m=...]$$

Then, we can typecheck:

$$o.m \Leftarrow \varsigma(Self<:Q, s':Q(Self), s:Q(Self))\ s'.recoup \ : \ Q$$

since s'.recoup has type Self. Moreover, the behavior obtained could be useful, and corresponds to storing the current object into the new object (like a "backup" operation).

We say that a method of the form $\varsigma(s:B(Self))\varsigma\langle Self,s\rangle$, in the context of a ς-object of the form $\varsigma(Self<:B=B)[...]$, is a recoup method. Intuitively, recoup allows us to recover a "parametric self " s'.recoup, which equals o but has type Self<:Q and not just type Q. This technique is particularly useful after an override on a value of type Self<:Q, because the result of the override only has type Q.

In general, if B has the form $\varsigma(Self^+)[recoup:Self, ...]$ then we can write useful polymorphic functions of type $\forall(Self<:B)\ B(Self)\rightarrow Self$ that are not available without recoup, such as:

$$g : \forall(Self<:Q)\ Q(Self)\rightarrow Self \triangleq$$
$$\lambda(Self<:Q)\ \lambda(s:Q(Self))\ (s.m:=s.recoup).recoup$$

Such a function is sufficiently parametric to be used in overrides from the outside, as in:

$$o.m \Leftarrow \varsigma(Self<:Q, s':Q(Self), s:Q(Self))\ g(Self)(s)$$

More generally, if Q' is a subtype of Q and o' has type Q', we can write:

$$o'.m \Leftarrow \varsigma(Self<:Q', s':Q'(Self), s:Q'(Self))\ g(Self)(s)$$

Thus, the function g, which may have been written with the type Q in mind, can be used for subtypes of Q. This is an instance of code reuse, in line with traditional object-oriented programming: the function that implements a method for a type can be reused (inherited) for implementing a method for a subtype.

The technique just described gives the correct result only as long as recoup is bound to $\varsigma(s:Q(Self))\varsigma\langle Self,s\rangle$. Otherwise, the operational behavior is not the expected one. The correctness of typing, on the other hand, does not depend on the recoup invariant.

An invariant of this kind is, we believe, perfectly acceptable for a programming language: recoup would be a distinguished component that is appropriately initialized and that cannot be overridden. Even without language support, we may be disciplined enough to preserve the recoup invariant, and thus we may solve the problem of overriding methods with result type Self.

6. Related Work

We finish with some comparisons with the most closely related work [6, 17], also discussed in [2]. We have fixed-size objects, and support subsumption by using a single subtyping relation. Mitchell *et al.* do not support subsumption but allow object extension; Bruce formalizes two distinct subtyping relations. Like Mitchell *et al.* and unlike Bruce, we do not distinguish between objects and object generators, and we allow the overriding of proper methods in objects. Many common examples can be expressed in all these systems.

Mitchell *et al.* and Bruce present systems with primitive objects and with a built-in Self type. In contrast we have a full second-order system where Self is obtained by an encoding. The rules for Self are similar in all these systems. The rules are always complex, but ours are derivable from those for elementary objects without Self. Hence, we may claim some success in explaining Self.

Acknowledgments

John Lamping prompted us to think about encoding covariant components. Gordon Plotkin suggested we should have a system with the Self quantifier as the only second-order construct.

Appendix

The following appendices list our rules. Appendices A, C, and D contain primitive rules, while appendices B and E contain derived rules. Appendix A contains rules for first-order objects; appendix C contains other typing fragments, and appendix D contains the corresponding equational fragments. Appendix B concerns ς-objects ; appendix E concerns the Self quantifier ς.

Appendix A: Simple-Objects Fragments

These are the typing and equational rules for simple objects.

$\Delta_{\mathbf{Ob}}$

(Type Object) (l_i distinct)

$$\frac{E \vdash B_i \quad \forall i \in 1..n}{E \vdash [l_i : B_i{}^{\,i \in 1..n}]}$$

(Val Object) (where $A \equiv [l_i : B_i{}^{\,i \in 1..n}]$)

$$\frac{E, x_i : A \vdash b_i : B_i \quad \forall i \in 1..n}{E \vdash [l_i = \varsigma(x_i : A) b_i{}^{\,i \in 1..n}] : A}$$

(Val Select)

$$\frac{E \vdash a : [l_i : B_i{}^{\,i \in 1..n}] \quad j \in 1..n}{E \vdash a.l_j : B_j}$$

(Val Override) (where $A \equiv [l_i : B_i{}^{\,i \in 1..n}]$)

$$\frac{E \vdash a : A \quad E, x : A \vdash b : B_j \quad j \in 1..n}{E \vdash a.l_j \Leftarrow \varsigma(x:A) b : A}$$

$\Delta_{\mathbf{<:Ob}}$

(Sub Object) (l_i distinct)

$$\frac{E \vdash B_i \quad \forall i \in 1..n+m}{E \vdash [l_i : B_i{}^{\,i \in 1..n+m}] <: [l_i : B_i{}^{\,i \in 1..n}]}$$

$\Delta_{\mathbf{=Ob}}$

(Eq Object) (where $A \equiv [l_i : B_i{}^{\,i \in 1..n}]$)

$$\frac{E, x_i : A \vdash b_i \leftrightarrow b_i' : B_i \quad \forall i \in 1..n}{E \vdash [l_i = \varsigma(x_i : A) b_i{}^{\,i \in 1..n}] \leftrightarrow [l_i = \varsigma(x_i : A) b_i'{}^{\,i \in 1..n}] : A}$$

(Eq Select)

$$\frac{E \vdash a \leftrightarrow a' : [l_i : B_i{}^{\,i \in 1..n}] \quad j \in 1..n}{E \vdash a.l_j \leftrightarrow a'.l_j : B_j}$$

(Eq Override) (where $A \equiv [l_i : B_i{}^{\,i \in 1..n}]$)

$$\frac{E \vdash a \leftrightarrow a' : A \quad E, x : A \vdash b \leftrightarrow b' : B_j \quad j \in 1..n}{E \vdash a.l_j \Leftarrow \varsigma(x:A) b \leftrightarrow a'.l_j \Leftarrow \varsigma(x:A) b' : A}$$

(Eval Select) (where $A \equiv [l_i : B_i{}^{\,i \in 1..n}]$, $a \equiv [l_i = \varsigma(x_i : A') b_i{}^{\,i \in 1..n+m}]$)

$$\frac{E \vdash a : A \quad j \in 1..n}{E \vdash a.l_j \leftrightarrow b_j\{x_j \leftarrow a\} : B_j}$$

(Eval Override) (where $A \equiv [l_i:B_i\ ^{i\in 1..n}]$, $a \equiv [l_i = \varsigma(x_i:A')b_i\ ^{i\in 1..n+m}]$)

$$\frac{E \vdash a : A \qquad E, x:A \vdash b : B_j \quad j\in 1..n}{E \vdash a.l_j \Leftarrow \varsigma(x:A)b \ \leftrightarrow\ [l_j = \varsigma(x:A')b,\ l_i = \varsigma(x_i:A')b_i\ ^{i\in(1..n+m)-\{j\}}] : A}$$

$\Delta_{=<:\mathbf{Ob}}$

(Eq Sub Object) (where $A \equiv [l_i:B_i\ ^{i\in 1..n}]$, $A' \equiv [l_i:B_i\ ^{i\in 1..n+m}]$)

$$\frac{E, x_i:A \vdash b_i : B_i \quad \forall i\in 1..n \qquad E, x_j:A' \vdash b_j : B_j \quad \forall j\in n+1..n+m}{E \vdash [l_i = \varsigma(x_i:A)b_i\ ^{i\in 1..n}] \ \leftrightarrow\ [l_i = \varsigma(x_i:A')b_i\ ^{i\in 1..n+m}] : A}$$

Appendix B: ς-Objects Fragments (Derived Rules)

These are derived rules for the combination of simple objects and the Self quantifier.

$\Delta_{\varsigma+}$

(Type ςObject)

$$\frac{E, X<:Top \vdash B_i\{X^+\} \quad \forall i \in 1..n}{E \vdash \varsigma(X^+)[l_i:B_i\{X\}^{\,i\in 1..n}]}$$

(Sub ςObject) (l_i distinct)

$$\frac{E, X<:Top \vdash B_i\{X^+\} \quad \forall i \in 1..n+m}{E \vdash \varsigma(X^+)[l_i:B_i\{X\}^{\,i\in 1..n+m}] <: \varsigma(X^+)[l_i:B_i\{X\}^{\,i\in 1..n}]}$$

(Val ςObject) (where $A \equiv \varsigma(X^+)[l_i:B_i\{X\}^{\,i\in 1..n}]$)

$$\frac{E \vdash C<:A \quad E \vdash b\{C\} : A\{C\}}{E \vdash \varsigma(Y<:A=C)\, b\{Y\} : A}$$

(Val ςSelect) (where $A \equiv \varsigma(X^+)[l_i:B_i\{X\}^{\,i\in 1..n}]$)

$$\frac{E \vdash a : A \quad j \in 1..n}{E \vdash a.l_j : B_j\{A\}}$$

(Val ςOverride) (where $A \equiv \varsigma(X^+)[l_i:B_i\{X\}^{\,i\in 1..n}]$)

$$\frac{E \vdash a : A \quad E, Y<:A, y:A\{Y\}, x:A\{Y\} \vdash b : B_j\{Y^+\} \quad j \in 1..n}{E \vdash a.l_j \Leftarrow \varsigma(Y<:A, y:A\{Y\}, x:A\{Y\})b : A}$$

$\Delta_{=\varsigma+}$

(Eq ςObject) (where $A \equiv \varsigma(X^+)[l_i:B_i\{X\}^{\,i\in 1..n}]$, $A' \equiv \varsigma(X^+)[l_i:B_i\{X\}^{\,i\in 1..n+m}]$)

$$\frac{E \vdash C<:A' \quad E \vdash b\{C\} \leftrightarrow b'\{C\} : A'\{C\}}{E \vdash \varsigma(Y<:A=C)b\{Y\} \;\leftrightarrow\; \varsigma(Y<:A'=C)b'\{Y\} : A}$$

(Eq Sub ςObject) (where $A \equiv \varsigma(X^+)[l_i:B_i\{X\}^{\,i\in 1..n}]$, $A' \equiv \varsigma(X^+)[l_i:B_i\{X\}^{\,i\in 1..n+m}]$)

$$\frac{E \vdash C<:A' \quad E, x_i:A\{C\} \vdash b_i\{C\} : B_i\{C\}\ \forall i \in 1..n \quad E, x_j:A'\{C\} \vdash b_j\{C\} : B_j\{C\}\ \forall j \in n+1..n+m}{E \vdash \varsigma(Y<:A=C)[l_i=\varsigma(x_i:A\{Y\})b_i\{Y\}^{\,i\in 1..n}] \;\leftrightarrow\; \varsigma(Y<:A'=C)[l_i=\varsigma(x_i:A'\{Y\})b_i\{Y\}^{\,i\in 1..n+m}] : A}$$

(Eq ςSelect) (where $A \equiv \varsigma(X^+)[l_i:B_i\{X\}^{\,i\in 1..n}]$)

$$\frac{E \vdash a \leftrightarrow a' : A \quad j \in 1..n}{E \vdash a.l_j \;\leftrightarrow\; a'.l_j : B_j\{A\}}$$

(Eq ςOverride) (where $A \equiv \varsigma(X^+)[l_i:B_i\{X\}^{\,i\in 1..n}]$)

$$\frac{E \vdash a \leftrightarrow a' : A \quad E, Y<:A, y:A\{Y\}, x:A\{Y\} \vdash b \leftrightarrow b' : B_j\{Y^+\} \quad j \in 1..n}{E \vdash a.l_j \Leftarrow \varsigma(Y<:A, y:A\{Y\}, x:A\{Y\})b \;\leftrightarrow\; a'.l_j \Leftarrow \varsigma(Y<:A, y:A\{Y\}, x:A\{Y\})b' : A}$$

(Eval ςSelect) (where $A \equiv \varsigma(X^+)[l_i:B_i\{X\}^{\,i\in 1..n}]$, $c \equiv \varsigma(Z<:A=C)a\{Z\}$)

$$\frac{E \vdash c : A \quad j \in 1..n}{E \vdash c.l_j \;\leftrightarrow\; a\{C\}.l_j : B_j\{A\}}$$

(Eval ς Override) (where $A \equiv \varsigma(X^+)[l_i:B_i\{X\}^{\,i\in 1..n}]$, $c \equiv \varsigma(Z<:A=C)a\{Z\}$)

$$\frac{E \vdash c : A \qquad E,\ Y<:A,\ y:A\{Y\},\ x:A\{Y\} \vdash b\{Y,y,x\} : B_j\{Y^+\} \qquad j\in 1..n}{E \vdash c.l_j \Leftarrow \varsigma(Y<:A,y:A(Y),x:A(Y))b\{Y,y,x\} \leftrightarrow \varsigma(Z<:A=C)\ a\{Z\}.l_j \Leftarrow \varsigma(x:A\{Z\})b\{Z,a\{Z\},x\} : A}$$

Appendix C: Other Typing Fragments

These are typing rules for standard constructs such as function types and recursive types.

$\Delta_{\mathbf{x}}$

(Env ∅) $\dfrac{}{\emptyset \vdash \diamond}$ (Env x) $\dfrac{E \vdash A \quad x \notin dom(E)}{E, x{:}A \vdash \diamond}$ (Val x) $\dfrac{E' \vdash \diamond}{E', x{:}A, E'' \vdash x{:}A}$

$\Delta_{\mathbf{K}}$

(Type Const) $\dfrac{E \vdash \diamond}{E \vdash K}$

$\Delta_{\rightarrow}$

(Type Arrow) $\dfrac{E \vdash A \quad E \vdash B}{E \vdash A \rightarrow B}$ (Val Fun) $\dfrac{E, x{:}A \vdash b : B}{E \vdash \lambda(x{:}A)b : A \rightarrow B}$ (Val Appl) $\dfrac{E \vdash b : A \rightarrow B \quad E \vdash a : A}{E \vdash b(a) : B}$

$\Delta_{<:}$

(Sub Refl) $\dfrac{E \vdash A}{E \vdash A <: A}$ (Sub Trans) $\dfrac{E \vdash A <: B \quad E \vdash B <: C}{E \vdash A <: C}$ (Val Subsumption) $\dfrac{E \vdash a : A \quad E \vdash A <: B}{E \vdash a : B}$

(Type Top) $\dfrac{E \vdash \diamond}{E \vdash Top}$ (Sub Top) $\dfrac{E \vdash A}{E \vdash A <: Top}$

$\Delta_{<:\rightarrow}$

(Sub Arrow) $\dfrac{E \vdash A' <: A \quad E \vdash B <: B'}{E \vdash A \rightarrow B <: A' \rightarrow B'}$

$\Delta_{<:\mathbf{X}}$

(Env X<:) $\dfrac{E \vdash A \quad X \notin dom(E)}{E, X{<:}A \vdash \diamond}$ (Type X<:) $\dfrac{E' \vdash \diamond}{E', X{<:}A, E'' \vdash X}$ (Sub X) $\dfrac{E' \vdash \diamond}{E', X{<:}A, E'' \vdash X{<:}A}$

$\Delta_{<:\mu}$

(Type Rec<:)

$$\frac{E, X<:Top \vdash A}{E \vdash \mu(X)A}$$

(Sub Rec)

$$\frac{E \vdash \mu(X)A \quad E \vdash \mu(Y)B \quad E, Y<:Top, X<:Y \vdash A<:B}{E \vdash \mu(X)A <: \mu(Y)B}$$

(Val Fold)

$$\frac{E \vdash a : A\{X \leftarrow \mu(X)A\}}{E \vdash fold(\mu(X)A, a) : \mu(X)A}$$

(Val Unfold)

$$\frac{E \vdash a : \mu(X)A}{E \vdash unfold(a) : A\{X \leftarrow \mu(X)A\}A}$$

$\Delta_{<:\forall}$

(Type All<:)

$$\frac{E, X<:A \vdash B}{E \vdash \forall(X<:A)B}$$

(Sub All)

$$\frac{E \vdash A' <: A \quad E, X<:A' \vdash B <: B'}{E \vdash \forall(X<:A)B <: \forall(X<:A')B'}$$

(Val Fun2<:)

$$\frac{E, X<:A \vdash b : B}{E \vdash \lambda(X<:A)b : \forall(X<:A)B}$$

(Val Appl2<:)

$$\frac{E \vdash b : \forall(X<:A)B\{X\} \quad E \vdash A'<:A}{E \vdash b(A') : B\{A'\}}$$

$\Delta_{<:\exists}$

(Type Exists<:)

$$\frac{E, X<:A \vdash B}{E \vdash \exists(X<:A)B}$$

(Sub Exists)

$$\frac{E \vdash A <: A' \quad E, X<:A \vdash B <: B'}{E \vdash \exists(X<:A)B <: \exists(X<:A')B'}$$

(Val Pack<:)

$$\frac{E \vdash C <: A \quad E \vdash b\{C\} : B\{C\}}{E \vdash (pack\ X<:A=C, b\{X\}:B\{X\}) : \exists(X<:A)B\{X\}}$$

(Val Open<:)

$$\frac{E \vdash c : \exists(X<:A)B \quad E \vdash D \quad E, X<:A, x:B \vdash d : D}{E \vdash (open\ c\ as\ X<:A,x:B\ in\ d:D) : D}$$

Appendix D: Other Equational Fragments

These are equational rules for standard constructs; corresponding to the rules of appendix C.

$\Delta_{=}$

(Eq Symm)

$$\frac{E \vdash a \leftrightarrow b : A}{E \vdash b \leftrightarrow a : A}$$

(Eq Trans)

$$\frac{E \vdash a \leftrightarrow b : A \quad E \vdash b \leftrightarrow c : A}{E \vdash a \leftrightarrow c : A}$$

$\Delta_{=x}$

(Eq x)

$$\frac{E', x{:}A, E'' \vdash \diamond}{E', x{:}A, E'' \vdash x \leftrightarrow x : A}$$

$\Delta_{=\rightarrow}$

(Eq Fun)

$$\frac{E, x{:}A \vdash b \leftrightarrow b' : B}{E \vdash \lambda(x{:}A)b \leftrightarrow \lambda(x{:}A)b' : A \rightarrow B}$$

(Eq Appl)

$$\frac{E \vdash b \leftrightarrow b' : A \rightarrow B \quad E \vdash a \leftrightarrow a' : A}{E \vdash b(a) \leftrightarrow b'(a') : B}$$

(Eval Beta)

$$\frac{E \vdash \lambda(x{:}A)b : A \rightarrow B \quad E \vdash a : A}{E \vdash (\lambda(x{:}A)b)(a) \leftrightarrow b\{x \leftarrow a\} : B}$$

(Eval Eta)

$$\frac{E \vdash b : A \rightarrow B \quad x \notin dom(E)}{E \vdash \lambda(x{:}A)b(x) \leftrightarrow b : A \rightarrow B}$$

$\Delta_{=<:}$

(Eq Subsumption)

$$\frac{E \vdash a \leftrightarrow a' : A \quad E \vdash A <: B}{E \vdash a \leftrightarrow a' : B}$$

(Eq Top)

$$\frac{E \vdash a : A \quad E \vdash b : B}{E \vdash a \leftrightarrow b : Top}$$

$\Delta_{=<:\mu}$

(Eq Fold)

$$\frac{E \vdash a \leftrightarrow a' : A\{X \leftarrow \mu(X)A\}}{E \vdash fold(\mu(X)A, a) \leftrightarrow fold(\mu(X)A, a') : \mu(X)A}$$

(Eq Unfold)

$$\frac{E \vdash a \leftrightarrow a' : \mu(X)A}{E \vdash unfold(a) \leftrightarrow unfold(a') : A\{X \leftarrow \mu(X)A\}}$$

(Eval Fold)

$$\frac{E \vdash a : \mu(X)A}{E \vdash fold(\mu(X)A, unfold(a)) \leftrightarrow a : \mu(X)A}$$

(Eval Unfold)

$$\frac{E \vdash a : A\{X \leftarrow \mu(X)A\}}{E \vdash unfold(fold(\mu(X)A, a)) \leftrightarrow a : A\{X \leftarrow \mu(X)A\}}$$

$\Delta_{=<:\forall}$

(Eq Fun2<:)

$$\frac{E, X{<:}A \vdash b \leftrightarrow b' : B}{E \vdash \lambda(X{<:}A)b \leftrightarrow \lambda(X{<:}A)b' : \forall(X{<:}A)B}$$

(Eq Appl2<:)

$$\frac{E \vdash b \leftrightarrow b' : \forall(X{<:}A)B\{X\} \quad E \vdash A' <: A}{E \vdash b(A') \leftrightarrow b'(A') : B\{A'\}}$$

(Eval Beta2<:)

$$\frac{E \vdash \lambda(X<:A)b : \forall(X<:A)B \quad E \vdash C <: A}{E \vdash (\lambda(X<:A)b)(C) \leftrightarrow b\{X \leftarrow C\} : B\{X \leftarrow C\}}$$

(Eval Eta2<:)

$$\frac{E \vdash b : \forall(X<:A)B \quad X \notin dom(E)}{E \vdash \lambda(X<:A)b(X) \leftrightarrow b : \forall(X<:A)B}$$

$\Delta_{=<:\exists}$

(Eq Pack<:)

$$\frac{E \vdash C <: A' \quad E \vdash A' <: A \quad E, X<:A' \vdash B'\{X\} <: B\{X\} \quad E \vdash b\{C\} \leftrightarrow b'\{C\} : B'\{C\}}{E \vdash (\text{pack } X<:A=C, b\{X\}:B\{X\}) \leftrightarrow (\text{pack } X<:A'=C, b'\{X\}:B'\{X\}) : \exists(X<:A)B\{X\}}$$

(Eq Open<:)

$$\frac{E \vdash c \leftrightarrow c' : \exists(X<:A)B \quad E \vdash D \quad E, X<:A, x:B \vdash d \leftrightarrow d' : D}{E \vdash (\text{open } c \text{ as } X<:A,x:B \text{ in } d:D) \leftrightarrow (\text{open } c' \text{ as } X<:A,x:B \text{ in } d':D) : D}$$

(Eval Unpack<:) (where $c \equiv \text{pack } X<:A=C, b\{X\}:B\{X\}$)

$$\frac{E \vdash c : \exists(X<:A)B\{X\} \quad E \vdash D \quad E, X<:A, x:B\{X\} \vdash d\{X,x\} : D}{E \vdash (\text{open } c \text{ as } X<:A,x:B\{X\} \text{ in } d\{X,x\}:D) \leftrightarrow d\{C,b\{C\}\} : D}$$

(Eval Repack<:)

$$\frac{E \vdash b : \exists(X<:A)B\{X\} \quad E, y:\exists(X<:A)B\{X\} \vdash d\{y\} : D}{E \vdash (\text{open } b \text{ as } X<:A,x:B\{X\} \text{ in } d\{\text{pack } X'<:A=X, x:B\{X'\}\}:D) \leftrightarrow d\{b\} : D}$$

Appendix E: The ς Ob Calculus

ς Ob is our minimal second-order object calculus. It is obtained by combining the rules for object types (appendix A) with the Self quantifier (section 3.4) taken as a primitive, plus some general rules (appendix C and D). The rules for ς-objects (appendix B) are derivable from the ones shown here.

A,B ::= X | Top | $[l_i:B_i\ ^{i\in 1..n}]$ | ς(X)B

a,b ::= x | $[l_i=\varsigma(x_i:A)b_i\ ^{i\in 1..n}]$ | a.l | $a.l \Leftarrow \varsigma(x:A)b$ | ς(X<:A=B)b | use a as X<:A, y:B in b:D

(Env ∅)
$$\frac{}{\emptyset \vdash \diamond}$$

(Env x)
$$\frac{E \vdash A \qquad x \notin dom(E)}{E, x:A \vdash \diamond}$$

(Env X<:)
$$\frac{E \vdash A \qquad X \notin dom(E)}{E, X<:A \vdash \diamond}$$

(Type X<:)
$$\frac{E', X<:A, E'' \vdash \diamond}{E', X<:A, E'' \vdash X}$$

(Type Top)
$$\frac{E \vdash \diamond}{E \vdash Top}$$

(Type Object) (l_i distinct)
$$\frac{E \vdash B_i \qquad \forall i \in 1..n}{E \vdash [l_i:B_i\ ^{i\in 1..n}]}$$

(Type Self)
$$\frac{E, X<:Top \vdash X}{E \vdash \varsigma(X)B}$$

(Sub Refl)
$$\frac{E \vdash A}{E \vdash A <: A}$$

(Sub Trans)
$$\frac{E \vdash A <: B \qquad E \vdash B <: C}{E \vdash A <: C}$$

(Sub X)
$$\frac{E', X<:A, E'' \vdash \diamond}{E', X<:A, E'' \vdash X<:A}$$

(Sub Top)
$$\frac{E \vdash A}{E \vdash A <: Top}$$

(Sub Object) (l_i distinct)
$$\frac{E \vdash B_i \qquad \forall i \in 1..n+m}{E \vdash [l_i:B_i\ ^{i\in 1..n+m}] <: [l_i:B_i\ ^{i\in 1..n}]}$$

(Sub Self)
$$\frac{E, X<:Top \vdash B <: B'}{E \vdash \varsigma(X)B <: \varsigma(X)B'}$$

(Val Subsumption)
$$\frac{E \vdash a : A \qquad E \vdash A <: B}{E \vdash a : B}$$

(Val x)
$$\frac{E', x:A, E'' \vdash \diamond}{E', x:A, E'' \vdash x:A}$$

(Val Object) (where $A \equiv [l_i:B_i\ ^{i\in 1..n}]$)
$$\frac{E, x_i:A \vdash b_i : B_i \qquad \forall i \in 1..n}{E \vdash [l_i=\varsigma(x_i:A)b_i\ ^{i\in 1..n}] : A}$$

(Val Select)
$$\frac{E \vdash a : [l_i:B_i\ ^{i\in 1..n}] \qquad j \in 1..n}{E \vdash a.l_j : B_j}$$

(Val Override) (where $A \equiv [l_i:B_i\ ^{i\in 1..n}]$)
$$\frac{E \vdash a : A \qquad E, x:A \vdash b : B_j \qquad j \in 1..n}{E \vdash a.l_j \Leftarrow \varsigma(x:A)b : A}$$

(Val Self) (where $A \equiv \varsigma(X)B\{X\}$)
$$\frac{E \vdash C <: A \qquad E \vdash b\{C\} : B\{C\}}{E \vdash \varsigma(Y<:A=C)\, b\{Y\} : A}$$

(Val Use) (where $A \equiv \varsigma(X)B\{X\}$)
$$\frac{E \vdash c : A \qquad E \vdash D \qquad E, Y<:A, y:B\{Y\} \vdash d : D}{E \vdash (use\ c\ as\ Y<:A, y:B\{Y\}\ in\ d:D) : D}$$

(Eq Symm)
$$\frac{E \vdash a \leftrightarrow b : A}{E \vdash b \leftrightarrow a : A}$$

(Eq Trans)
$$\frac{E \vdash a \leftrightarrow b : A \qquad E \vdash b \leftrightarrow c : A}{E \vdash a \leftrightarrow c : A}$$

(Eq Subsumption)
$$\frac{E \vdash a \leftrightarrow a' : A \qquad E \vdash A <: B}{E \vdash a \leftrightarrow a' : B}$$

(Eq x)
$$\frac{E', x{:}A, E'' \vdash \diamond}{E', x{:}A, E'' \vdash x \leftrightarrow x : A}$$

(Eq Top)
$$\frac{E \vdash a{:}A \qquad E \vdash b{:}B}{E \vdash a \leftrightarrow b : Top}$$

(Eq Object) (where $A \equiv [l_i{:}B_i\ ^{i \in 1..n}]$)
$$\frac{E, x_i{:}A \vdash b_i \leftrightarrow b_i' : B_i \quad \forall i \in 1..n}{E \vdash [l_i = \varsigma(x_i{:}A) b_i\ ^{i \in 1..n}] \leftrightarrow [l_i = \varsigma(x_i{:}A) b_i'\ ^{i \in 1..n}] : A}$$

(Eq Sub Object) (where $A \equiv [l_i{:}B_i\ ^{i \in 1..n}]$, $A' \equiv [l_i{:}B_i\ ^{i \in 1..n+m}]$)
$$\frac{E, x_i{:}A \vdash b_i : B_i \quad \forall i \in 1..n \qquad E, x_j{:}A' \vdash b_j : B_j \quad \forall j \in n+1..n+m}{E \vdash [l_i = \varsigma(x_i{:}A) b_i\ ^{i \in 1..n}] \leftrightarrow [l_i = \varsigma(x_i{:}A') b_i\ ^{i \in 1..n+m}] : A}$$

(Eq Select)
$$\frac{E \vdash a \leftrightarrow a' : [l_i{:}B_i\ ^{i \in 1..n}] \qquad j \in 1..n}{E \vdash a.l_j \leftrightarrow a'.l_j : B_j}$$

(Eq Override) (where $A \equiv [l_i{:}B_i\ ^{i \in 1..n}]$)
$$\frac{E \vdash a \leftrightarrow a' : A \qquad E, x{:}A \vdash b \leftrightarrow b' : B_j \qquad j \in 1..n}{E \vdash a.l_j \Leftarrow \varsigma(x{:}A) b \leftrightarrow a'.l_j \Leftarrow \varsigma(x{:}A) b' : A}$$

(Eq Self) (where $A \equiv \varsigma(X)B\{X\}$, $A' \equiv \varsigma(X)B'\{X\}$)
$$\frac{E \vdash C <: A' \qquad E, X <: Top \vdash B'\{X\} <: B\{X\} \qquad E \vdash b\{C\} \leftrightarrow b'\{C\} : B'\{C\}}{E \vdash \varsigma(Y <: A = C) b\{Y\} \leftrightarrow \varsigma(Y <: A' = C) b'\{Y\} : A}$$

(Eq Use) (where $A \equiv \varsigma(X)B\{X\}$)
$$\frac{E \vdash c \leftrightarrow c' : A \qquad E \vdash D \qquad E, Y <: A, y{:}B\{Y\} \vdash d \leftrightarrow d' : D}{E \vdash (use\ c\ as\ Y <: A, y{:}B\{Y\}\ in\ d{:}D) \leftrightarrow (use\ c'\ as\ Y <: A, y{:}B\{Y\}\ in\ d'{:}D) : D}$$

(Eval Select) (where $A \equiv [l_i{:}B_i\ ^{i \in 1..n}]$, $a \equiv [l_i = \varsigma(x_i{:}A') b_i\ ^{i \in 1..n+m}]$)
$$\frac{E \vdash a : A \qquad j \in 1..n}{E \vdash a.l_j \leftrightarrow b_j\{x_j \leftarrow a\} : B_j}$$

(Eval Override) (where $A \equiv [l_i{:}B_i\ ^{i \in 1..n}]$, $a \equiv [l_i = \varsigma(x_i{:}A') b_i\ ^{i \in 1..n+m}]$)
$$\frac{E \vdash a : A \qquad E, x{:}A \vdash b : B_j \qquad j \in 1..n}{E \vdash a.l_j \Leftarrow \varsigma(x{:}A) b \leftrightarrow [l_j = \varsigma(x{:}A') b,\ l_i = \varsigma(x_i{:}A') b_i\ ^{i \in (1..n+m) - \{j\}}] : A}$$

(Eval Unself) (where $A \equiv \varsigma(X)B\{X\}$, $c \equiv \varsigma(Z <: A = C) b\{Z\}$)
$$\frac{E \vdash c : A \qquad E \vdash D \qquad E, Y <: A, y{:}B\{Y\} \vdash d\{Y,y\} : D}{E \vdash (use\ c\ as\ Y <: A, y{:}B\{Y\}\ in\ d\{Y,y\}{:}D) \leftrightarrow d\{C, b\{C\}\} : D}$$

(Eval Reself) (where $A \equiv \varsigma(X)B\{X\}$)
$$\frac{E \vdash b : A \qquad E, y{:}A \vdash d\{y\} : D}{E \vdash (use\ b\ as\ Y <: A, y{:}B\{Y\}\ in\ d\{\varsigma(Y' <: A = Y) y\} : A) \leftrightarrow d\{b\} : D}$$

References

[1] Abadi, M. and L. Cardelli, **A semantics of object types**. *Proc. IEEE Symposium on Logic in Computer Science*, 332-341. 1994.

[2] Abadi, M. and L. Cardelli, **A theory of primitive objects: untyped and first-order systems**. *Proc. Theoretical Aspects of Computer Software*. Springer-Verlag. 1994.

[3] Abadi, M. and L. Cardelli, **An imperative object calculus**. *Proc. TAPSOFT'95*, 471-485. Springer-Verlag. 1995.

[4] Abadi, M. and L. Cardelli, **On subtyping and matching**. *Proc. ECOOP'95 (to appear)*. Springer-Verlag. 1995.

[5] Böhm, C. and A. Berarducci, **Automatic synthesis of typed λ-programs on term algebras**. *Theoretical Computer Science* **39**, 135-154. 1985.

[6] Bruce, K.B., **A paradigmatic object-oriented programming language: design, static typing and semantics**. *Journal of Functional Programming* **4**(2), 127-206. 1994.

[7] Bruce, K.B., L. Cardelli, G. Castagna, The Hopkins Objects Group, G.T. Leavens, and B. Pierce, **On binary methods**. TR95-08. Department of Computer Science, Iowa State University. 1995.

[8] Cardelli, L., **Extensible records in a pure calculus of subtyping**. In *Theoretical Aspects of Object-Oriented Programming*, C.A. Gunter and J.C. Mitchell, ed. MIT Press. 373-425. 1994.

[9] Cardelli, L., J.C. Mitchell, S. Martini, and A. Scedrov, **An extension of system F with subtyping**. *Proc. Theoretical Aspects of Computer Software*, 750-770. Lecture Notes in Computer Science 526. Springer-Verlag. 1991.

[10] Cardelli, L. and P. Wegner, **On understanding types, data abstraction and polymorphism**. *Computing Surveys* **17**(4), 471-522. 1985.

[11] Curien, P.-L. and G. Ghelli, **Coherence of subsumption, minimum typing and type-checking in $F_{\leq}$**. *Mathematical Structures in Computer Science* **2**(1), 55-91. 1992.

[12] Dahl, O. and K. Nygaard, **Simula, an Algol-based simulation language**. *Communications of the ACM* **9**, 671-678. 1966.

[13] Fisher, K. and J.C. Mitchell, **Notes on typed object-oriented programming**. *Proc. Theoretical Aspects of Computer Software*, 844-885. Springer-Verlag. 1994.

[14] Girard, J.-Y., **Interprétation fonctionelle et élimination des coupures dans l'arithmétique d'ordre supérieur**. Thèse de doctorat d'état, University of Paris. 1972.

[15] Girard, J.-Y., Y. Lafont, and P. Taylor, **Proofs and types**. Cambridge University Press. 1989.

[16] Meyer, B., **Object-oriented software construction**. Prentice Hall. 1988.

[17] Mitchell, J.C., F. Honsell, and K. Fisher, **A lambda calculus of objects and method specialization**. *Proc. 8th Annual IEEE Symposium on Logic in Computer Science*. 1993.

[18] Mitchell, J.C. and G.D. Plotkin, **Abstract types have existential type**. *Proc. 12th Annual ACM Symposium on Principles of Programming Languages*. 1985.

[19] Nelson, G., ed. **Systems programming with Modula-3**. Prentice Hall. 1991.

[20] Palsberg, J., **Efficient inference for object types**. *Proc. 9th Annual IEEE Symposium on Logic in Computer Science*, 186-195. 1994.

[21] Plotkin, G.D., M. Abadi, and L. Cardelli, **Subtyping and parametricity**. *Proc. IEEE Symposium on Logic in Computer Science*, 310-319. To appear. 1994.

[22] Reynolds, J.C., **Towards a theory of type structure**. *Proc. Colloquium sur la programmation*, 408-423. Lecture Notes in Computer Science 19. Springer-Verlag. 1974.

[23] Szypersky, C., S. Omohundro, and S. Murer, **Engineering a programming language: the type and class system of Sather**. TR-93-064. ICSI, Berkeley. 1993.

[24] Wadsworth, C., **Some unusual λ-calculus numeral systems.** In *To H.B. Curry: Essays on combinatory logic, lambda calculus and formalism*, J.P. Seldin and J.R. Hindley, ed. Academic Press. 1980.